Lab 9: Recursive Functions

Assigned: *Tuesday, December 1st, 2020*
Due: *Friday, December 11th, 2020*
Points: *100*

- There is NO MAKEUP for missed assignments.
- We are strict about enforcing the LATE POLICY for all assignments (see syllabus).

Introduction

The assignment for this week will utilize concepts you've learned regarding working with recursive functions in C++.

It is HIGHLY RECOMMENDED that you develop the algorithms for each program FIRST and THEN develop the C++ code for it.

Step 1: Getting Ready

First, open a terminal window and log into the correct machine. Change into your CS 16 directory, create a **lab09** directory and change into it. There are no skeleton files provided for you.

Step 2: Create and Edit Your C++ Files

This week, you will need to create **multiple files** for both exercises in this lab, described below. Each exercise is worth 50 points.

IMPORTANT NOTE: This lab will be also graded for use of comments and style. See below for details. In addition, we will take *major* points OFF if you use C++ instructions/libraries/code that either (a) was not covered in class, or (b) was found to be copied from outside sources (or each other) without proper citation.

selectionSort.cpp

You remember we covered the Selection Sort algorithm in **Lecture 11** in this class. You have to write a recursive version of this algorithm to be able to sort an array of integers into either ascending *or* descending order using the following idea: Place the smallest/largest element in the first position, then sort the rest of the array by a recursive call. Note: Simply taking the program from lecture and plugging in a recursive version of the function "**index_of_smallest"** will not suffice. The function to do the sorting (called "**sort**" here) ***must itself be recursive*** and not merely *use* a recursive function.

REQUIREMENTS, ASSUMPTIONS:

1. Be sure to utilize ONLY techniques we've covered in lecture. Do not use "special" arrays or pointers, do not use vectors, do not use built-in sorting functions, etc… You will get zero points if you do.

2. You are given **2** skeleton programs and **1** example final program: **headers.h**, **functions.cpp**, and **selectionSort.cpp**, respectively. You must complete the first 2 files and submit them. The 3rd file, **selectionSort.cpp**, is an example of a program running and testing an integer array with the sort algorithm. In this program, the array is read from an input file, very much like what we did in **Lab 4**'s *array.cpp* exercise.

3. In both skeleton files, there are comments in there as guidelines to help you finish this exercise. You should READ THEM CAREFULLY and remove them before you submit. Of course, you are still expected to place appropriate comments of your own, per the styling guidelines of this course.

4. There are FOUR functions that you must define for this exercise. Declare them in **headers.h** and define them in **functions.cpp**. The function requirements are detailed below:

    a. The function **`getArray`** is the SAME one we used in **Lab 4**'s *arrays.cpp* exercise. This function reads a text file that contains only integers and places them in an array. The size of the array and the name of the file are found in global variables already defined for you in the **headers.h** file. An input text file, **ArrayFile.txt**, is also provided for you to test out your final program.

    b. The function **`sort`** is the *recursive* function that you must design. It is based on the Selection Sort function **sort()** that I introduced to you in **Lecture 11**. You have to re-design it to be a recursive function. ***If you fail at meeting this requirement, you will get a zero on this entire exercise, even if your code passes the autograder***. This function calls 2 other functions (just like the example in **Lect.11** – but read the details below for more information) and it **MUST** have 4 arguments (in this order):
        i. a Boolean variable that holds the sorting direction choice "Descending Sort" (true) or "Ascending Sort" (false).
        ii. An integer array that needs sorting
        iii. An integer that indicates the size of the array (or sub-array) that needs sorting.
        iv. An integer that indicates the starting index of the array that needs sorting. This is set to zero at first by the calling routing (i.e. the **main** function that calls **sort**).

    c. The function **`find_index_of_swap`** is a *non-recursive* function that you must design. It is based on the Selection Sort function **index_of_smallest()** that I introduced to you in **Lecture 11**. You have to re-design it to work with the Descening/Ascending choice. This function **MUST** have 4 arguments (in this order):
        i. The aforementioned Boolean variable for sorting direction.
        ii. The integer array.
        iii. The integer for size of the array (or sub-array).
        iv. The integer that indicates the starting index of the array that needs sorting.

    d. The function **`swap_values`** is the SAME one we used in **Lecture 11**'s Selection Sort example.

A session should look like one of the following examples (including whitespace and formatting). The user input is shown bolded for your convenience here. This is what you would see if you ran your (correctly done) functions via the main function in **selectionSort.cpp**.

```
$ ./selectionSort
Original array:
23 22 -5 -21 21 0 -12 -55 91 123 1 5 3 6 9 12 -11 17 8 -1
Ascending (0) or Descending (1): 0
Sorted array:
-55 -21 -12 -11 -5 -1 0 1 3 5 6 8 9 12 17 21 22 23 91 123

$ ./selectionSort
Original array:
23 22 -5 -21 21 0 -12 -55 91 123 1 5 3 6 9 12 -11 17 8 -1
Ascending (0) or Descending (1): 1
Sorted array:
123 91 23 22 21 17 12 9 8 6 5 3 1 0 -1 -5 -11 -12 -21 -55
```

HINTS

Let's say you have this integer array and you want to sort it in ascending fashion:

23	22	-5	0	1	2

Per the selection sort algorithm, you first find the minimum value (-5) and then swap it with the first element (23, index of 0). So, your array becomes:

-5	22	23	0	1	2

Since the first element (index 0) is now in its correct place, you can recursively do the sort if you now ONLY consider your sub-array that starts at the next element (index 1). Then you re-do the recursion starting at next element after that (index 2), etc… until you are done!

palindrome.cpp

A *palindrome* is a word or phrase whose meaning may be interpreted the same way in either forward or reverse direction. Famous examples include “Amore, Roma”, “A man, a plan, a canal: Panama” and “No ‘x’ in ‘Nixon’”.

Write a ***recursive function*** called **`isPalindrome`** that returns a Boolean value of true if an input string is a palindrome and false if it is not. You can do this by checking if the first character equals the last character, and if so, make a recursive call with the input string minus the first and last characters. You will have to define a suitable stopping condition. ***If you fail at meeting the requirement of making this function a recursive one, you will get a zero on this entire exercise, even if your code passes the autograder.***

Important: the string may contain ANY character, like whitespaces, hash-tags, numbers, etc… You should only be checking if the alphabet characters constitute a palindrome (ignore the character case). An input string that has NO letters (or is empty) is considered to be a palindrome (nothing backwards is still nothing!)

Look at the **palindrome_skeleton.cpp** program provided and the **main()** function therein. It takes in a string as user input, then calls 2 functions (**`cleanUp`** and **`isPalindrome`**) and, on the basis of the answer it sees, it outputs the result.

You have to:

a. Figure out the definitions (and declarations!) of these 2 functions.
b. Again, the function **`isPalindrome`** HAS to be a recursive function.
c. You can create more than those 2 functions if you like, but you do not need to and likely will *not need to* either.
d. Make sure to rename the file **palindrome.cpp** before you submit it and to style the program correctly (including placing the appropriate number of comments).

Here are a few example sessions:
The program should print a string of text to the terminal before getting the inputs from the user. A session should look like one of the following examples (including whitespace and formatting):

```
$ ./palindrome
Enter sentence:
hello
It is not a palindrome.
$ ./palindrome
Enter sentence:
Madam, I'm Adam
It is a palindrome.
$ ./palindrome
Enter sentence:
MADAM!! I'M a d  am!?
It is a palindrome.
$ ./palindrome
Enter sentence:
Madam I'm ada
It is not a palindrome.
```

Here are some more example runs:

```
$ ./palindrome
Enter sentence:
A Santa dog lived as a devil God at NASA
It is a palindrome.
$ ./palindrome
Enter sentence:
...Maps, DNA, and spam...
It is a palindrome.
$ ./palindrome
Enter sentence:
As I pee, sir, I see Pisa!
It is a palindrome.
$ ./palindrome
Enter sentence:
Just as I suspected...
It is not a palindrome.
$ ./palindrome
Enter sentence:

It is a palindrome.
```

Notes:

- That last entry in the example above was just an empty string.
- Each example above is a separate run – there's no repeating loop.

You MUST use a recursive function to build this program and you may not use built-in C++ functions or any other techniques that we have NOT discussed in lecture.

HINTS

Let's take the example of the string: "Madam, I'm Adam", a famous palindrome… You start off by "cleaning it up" of all punctuation, white spaces, and convert upper-case letters to get: "madamimadam". Now you compare string[0] with string[last_position] (i.e. 'm' and 'm'), if they are equal, you continue.

m	a	d	a	m	i	m	a	d	a	m

Now you no longer need these end letters. You can ignore them:

a	d	a	m	i	m	a	d	a

Now, again, you compare string[0] with string[last_position] (i.e. 'a and 'a), if they are equal, you continue. Again, you can ignore these end letters (you can even discard them):

d	a	m	i	m	a	d

Now, again, you compare string[0] with string[last_position] (i.e. 'd and 'd), if they are equal, you continue…etc.. etc…

You keep doing this until there is nothing left to compare (think about how that is determined because this is your base-case). If, along the way, every comparison was equal, then you have a palindrome. If even ONE comparison was not equal, then you don't have a palindrome.

Step 3: Create a makefile and Compile the Codes with the make Command

In order to learn another way to manage our source codes and their compilations, we will first create a **makefile** and put in the usual g++ commands in it. Afterwards, whenever we want to compile our programs, the Linux command is a lot shorter – so this is a convenience.

Using your text editor, create a new file called **makefile** and enter the following into it:

```
all: selectionSort palindrome

selectionSort: selectionSort.cpp headers.h functions.cpp
	g++ -o selectionSort selectionSort.cpp -Wall -std=c++11

palindrome: palindrome.cpp
	g++ -o palindrome palindrome.cpp -Wall -std=c++11

clean:
	rm -f a.out *.o selectionSort palindrome
```

Then from the Linux prompt, you can do one of two things: either issue separate compile commands for each project, like so:

```
$ make selectionSort
```

Or, you can issue one command that will compile all the projects mentioned in the **makefile**, like so:

```
$ make
```

If the compilation is successful, you will not see any output from the compiler. You can then run your programs, for example:

```
$ ./selectionSort
```

If you encounter an error, use the compiler hints and examine the line in question. If the compiler message is not sufficient to identify the error, you can search online to see when the error occurs in general.

Remember to re-compile the relevant files after you make any changes to the C++ code.

Step 4: Submit using GRADESCOPE

BEFORE YOU SUBMIT YOUR FINAL VERSION FILES: Make sure that you:

a) **STYLIZE** your program appropriately to additionally make it easier on others reading your code. Apply ALL the style pointers I have posted on Piazza. This includes the use of comments.

This will be graded for an *extra 20 points* beyond the automatic grade you get from Gradescope.

b) **MEET THE REQUIREMENTS OF THIS LAB**. This means, if I said certain features or program aspects were *required* to be in your final code, then they must be there.

If these requirements are not met, you will lose MAJOR points on the exercise. This simulates a real-world situation (e.g. your customer wants certain features, so you must provide it – while I am not your customer, per se, I *am* giving you points for your efforts!)

Log into your account on **https://www.gradescope.com/** and navigate to our course site. Select this assignment. Then click on the "Submit" button on the bottom right corner to make a submission. You will be given the option of uploading files from your local machine or submitting the code that is in a GitHub repo. Choose the first option and follow the steps to **upload your 3 files** to Gradescope.

These files should be: **headers.h** and **functions.cpp** from the 1st exercise (you don't upload selectionSort.cpp since we will have our own based on the example we gave you) and **palindrome.cpp** from the 2nd exercise.

You may submit this lab multiple times. You should submit only after you test it on your computer (and CSIL) and are satisfied that the programs run correctly. The score of the last submission uploaded before the deadline will be used as your assignment grade.

Step 5: Done!

Once your submission receives a score of 100/100 on the autograder, you are done with this assignment. REMEMBER that there are 20 additional points that will be scored according to your proper use of comments and styling. The total will then be normalized to 100 points again (i.e. 120 score will be 100%) to grade this lab.

WE WILL BE CHECKING FOR PLAGIARISM – DO NOT COPY FROM OTHER STUDENTS OR FROM SOURCES ONLINE! USE PROPER CITATION OF CODE THAT YOU DID NOT WRITE! THE CONSEQUENCES WILL - AT MINIMUM - BE A ZERO ON THIS LAB AND POSSIBLY A ZERO IN THIS COURSE AND YOU WILL BE REPORTED TO THE UNIVERSITY.