Lab 8: Released Wednesday, week 9 and due Tuesday, week 10

Recursive Functions

Introduction

The assignment for this week will involve designing recursive functions.

We will also be grading for meeting requirements, using “class legal” code, and plagiarism. So, it is not enough for your lab to just pass the Gradescope autograder! Please read the instructions herein carefully.

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

Recall what environment you set up in lab 1. Navigate there, and then clone the Github repository. Go to the course Github (https://github.com/ucsb-cs16-1-u22) and make sure you can see a repository named Lab08-yourgithubusername. If you can see that repository, open it up in your browser and go to the green “Code” button near the upper right. Copy the SSH link for this repository (for more instructions on this step, and future steps involving Github, check out https://sites.cs.ucsb.edu/~maradowning/cs16/refs/gitbasics.html).

From here, run the following command in the folder you’d like your Lab08 folder to end up in:
`git clone <link you just copied>`

Check afterwards with `ls` to make sure the repository has been cloned properly, you should see a new folder named Lab08-yourgithubusername. You can run the following command to enter this directory and see the starter code files:
`cd Lab08-yourgithubusername`

Step 2: Create and Edit your C++ Programs

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: 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

Remember we covered the Selection Sort algorithm in lecture. 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

1. Be sure to utilize only techniques we've covered in lecture. 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.hpp, functions.cpp, and selectionSort.cpp, respectively. You must complete the first 2 files and submit them. The third 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 before you submit.
4. There are four functions that you must define for this exercise. Declare them in `headers.hpp` 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. 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 lecture—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 routine (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. You have to re-design it to work with the Descending/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 the lecture’s Selection Sort example. Feel free to copy it over directly.

A session should look like one of the following examples (including whitespace and formatting). This is what you would see if you ran your (correctly done) functions via the main function in selectionSort.cpp twice, choosing ascending first, and then descending.

```
$ ./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 the next element after that (index 2), etc., until you are done!

palindrome.cpp
A palindrome is a word or phrase whose letters are the same 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 do not make 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, punctuation, 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.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: 1. Figure out the definitions (and declarations!) of these 2 functions. 2. Again, the function `isPalindrome` must be a recursive function. 3. You can create more than those 2 functions if you like, but you do not need to.

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.
$ ./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:
Just as I suspected...
It is not a palindrome.
$ ./palindrome
Enter sentence:

It is a palindrome.
```

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]
```

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: Compile your Code

To compile your programs, you can run
`make selectionSort`
and
`make palindrome`

You can also just run
`make`
to compile both at the same time.

If you need to delete your executable files, running
`make clean`
will delete them

You can run the executable files as normal:
`./selectionSort`
`./palindrome`

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 after you make any changes to your C++ source code.

Step 4: Submit your Code

There is a more detailed (and more general) set of instructions on the course webpage, under Quick References: https://sites.cs.ucsb.edu/~maradowning/cs16/refs/gitbasics.html. If you are new to git/github, I suggest reading through this guide as well when you submit your assignment.

Once you have finished your code (or any time you want to save your code to Github and take a break), first run `git status` to see which files you have modified/created. From there, add any files you want to end up on Github by running the following command:

```
git add <filename>
```

For this lab, you will only need to add functions.cpp, headers.hpp, and palindrome.cpp

Please do not add any executable files you created by compiling your code.

Once you've added all the files, you will need to commit them. Run:

```
git commit -m "Your commit message here"
```

Please write a descriptive commit message within the quotes, something like "Finished Lab08" if you are done, or something more specific if you're pushing code partway through.

Finally, run:

```
git push
```

to push your changes to Github. Once you've run this command, make sure to go to your repository in your web browser and check that the updated files are there (you might have to reload).

Finally, once you're completely finished with the lab, fill out the Lab08 submission on Gradescope. It only asks you for a list of people you discussed the lab with (remember, discussion of the lab is ok but do not give classmates answers), the number of hours you spent, and has a checkbox to indicate that you're done. I will use this as an indication that your code is pushed to Github and I can begin grading.