Recursion Lab

Reading, Background, and General Instructions

zyBook through Chapter 10.

References given in the assignment.

Other references as you deem appropriate to develop the system.

Problems must be solved recursively. No global variables. No static variables in functions. All results are return values or modified arguments called by reference. Function arguments must be as specified in the instructions. If you are unsure about your solution, ask. Solutions that do not adhere to instructions and spirit of recursive solutions will be considered incorrect.

Part 1

Simple (or not-so-simple) recursive functions

Starting with the templates provided below, write a program that inputs one-character commands that instruct the program to run one of the recursive functions below. Use **switch** to select function. Output appropriate results. All I/O should be done in main. A correct executable is available: **/home/Csc112-S-F16/recur**. You must use a multi-file format. Implement your code using clion using separate header and c++ files for the functions. I.e., all functions can be declared and coded using one **.h** and one **.cpp** file. Files:

- **main.cpp**
- **Recur_Fun.h**
- **Recur_Fun.cpp**

Submit and test the three files in zyLab. Comments in the code should explain how your recursion works. Yes, these functions must be implemented recursively. That is, the functions may not *fake it* by a loop or other non-recursive method just to make it work. This is the requirement and spirit of the problem.

Recursive functions to implement. One-character command is in [] before the function name.

- **[R] str_rev** returns the reverse of the argument string. Parameter is passed by reference and constant. Input the string.
- **[M] list_max** returns the max of a non-negative, integer vector passed by reference and may be changed by function if needed. Input the integers. End when negative value. This recursive implementation can be tricky. Some thoughts. Base case is one element in the vector. Otherwise, return the max(list(0), list_max(tail(list))). tail(list) is the list with list(0) removed. You can also do the same thing by starting at the end of the list instead of the beginning, which would be more efficient.
- **[P] palin** returns a Boolean indicating whether or not the argument string is a palindrome. Parameter is constant passed by value. Input the string.
- **[F] find_sub** returns a Boolean indicating if the first argument string is a substring of the second. That is, the first argument occurs anywhere in the second. E.g., "to", "day, "oda", are all substrings of "today". Also, "today" would be considered a substring of itself. Parameters are constant pass by reference. Input the substring to find followed by the string to search.
- **[E] N**ot a function but is the command to end the program

```
// Partial code for main.cpp
// Students may use if they like.
#include "Recur_Fun.h"
using namespace std;
int main() {
   char    in_cmd;
   int     in_int, ret_int;
   bool    ret_bool;
   vector <int> int_vect;
   string  in_str_1, in_str_2, ret_str;

   cout << "Cmd: ";
   cin >> in_cmd;

   while (in_cmd != 'E') {
      switch (in_cmd) {
         case 'R':
            cout << "Str: " << endl;
            cin >> in_str_1;
            ret_str = str_rev(in_str_1);
            cout << ret_str << endl;
           break;
         case 'F':
            cout << "Substr: " << endl;
            cin >> in_str_1;
            cout << "Mainstr: " << endl;
            cin >> in_str_2;
            ret_bool  = find_sub(in_str_1, in_str_2);
            if (ret_bool) {
               cout << "True" << endl;
            }
            else {
               cout << "False" << endl;
            }
            break;
         default:
            cout << "Bad Cmd" << endl;
            break;

      }
      cout << "Cmd: " << endl;
      cin >> in_cmd;
   }
   return 0;
}
```

```
// Partial code for Recur_Fun.h
// Students may use if they like.
//
// Recursive Function Declarations
//
#include <iostream>
#include <vector>
#ifndef LAB_5_PART_0_RECUR_FUN_H
#define LAB_5_PART_0_RECUR_FUN_H

std::string str_rev(const std::string &arg_str);

#endif //LAB_5_PART_0_RECUR_FUN_H
```

Part 2

Tower of Hanoi (ToH). This recursive algorithm is elegant and demonstrates an effective and intuitive approach – once understood – to an interesting problem. You should first read about the problem. Notes and a starting code template are provided below. Submit your code to zyLab. Put all code in one file, so only **main.cpp** is needed. Declare the **move_discs** function before main and put the function code after main.

Solve the ToH problem recursively. By solve, this means to show all the moves to accomplish the task of moving the discs from the Tower A to the Tower B, obeying the rules of movement. Your particular solution should have three towers named: "**A**" "**B**" and "**C**".

The recursive function should be:

void move_discs(unsigned int num_discs, string tower_1, string tower_2, string tower_3, unsigned int &num_moves)

- **num_discs** – number of discs to move.
- **tower_1**, **tower_2**, **tower_3** – names of the three towers. You should move discs from **tower_1** to **tower_2** using **tower_3** as the intermediate.
- **num_moves** – keeps track of total number of moves.

Your input is the number of discs that start on **A** and must be moved to **B**. **C** can be used as needed. Only use one file, **main.cpp**. Declare **move_discs** before main and put the code for **move_discs** after **main**. The main should call:

move_discs(num_discs, "A", "B", "C", num_moves)

That is, move the discs indicated by **num_discs** from **A** to **B** using **C** as an intermediate tower (tower).

Your output should show each move taken, including which disc is moved. At the end, print the actual number of moves made. You must keep track of this and return in the referenced variable, **num_moves** . An example output is given below. You may want to try this with a physical example to see if and how it works. For our use, assume that disc #1 is top or smallest disc. The larger the disc number, the larger the disc.

```
Number of discs:
Move disc 1 from A to B
Move disc 2 from A to C
Move disc 1 from B to C
Move disc 3 from A to B
Move disc 1 from C to A
Move disc 2 from C to B
Move disc 1 from A to B
Number of moves: 7
```

Notes on the Tower of Hanoi Problem

You can find a description of the problem in various places.

- https://en.wikipedia.org/wiki/Tower_of_Hanoi - from here, you will also learn that the minimum number of moves required to solve a Tower of Hanoi puzzle is $(2^n - 1)$, where n is the number of discs. (Does that number look familiar? Is there a reason for this?)

- https://www.youtube.com/watch?v=bIgjzlumfsQ

- For more interested students, you can see the following link, http://mathworld.wolfram.com/TowerofHanoi.html

I like to think of the solution like this: You can show a base case. You can choose, 1 or 2 discs is good. Three if you like. I.e., you can find a solution for this. Once you have this, you can always move this number of disc to the temp disc. Thus, if you have one more, move it to the **B** disc, then move the ones from the temp to the **B**. See the recursion? The only change is that the intermediate source and destination towers may change, so you have to keep track of that in the recursive calls. Now you can do it for n+1 discs.

For those of you who have had Math 117 or have studied induction, you can use similar thinking for this problem, or for recursion in general. Base case is 1, and it works by running through the algorithm. Assume it works for the n^{th} case. Thus, at n-1, n-1 discs are on **C** tower (the temporary tower), disc n is on the **A** tower, and the program moves disc n to **B** and then n-1 the n-1 discs to **B**. Show the algorithm works for n+1 case. If it works for n, then n discs are on the **C** tower, program moves disk n+1 to **B** (the destination tower) then the algorithm successfully moves the n discs to the **B** since the algorithm works for n.

Starting template

```
int main() {
   unsigned int n_discs, num_moves=0;
   cout << "Number of discs: " << endl;
   cin >> n_discs;

   // Call to move_discs() goes here.

   cout << "Number of moves: " << num_moves << endl;
   return (0);
}

// Recursive function, move_discs(), goes here.
```

Part 3

Permutations of a string. Write a recursive solution to the permutations of a string problem. For this problem, the input is a string and the output is all permutations of the string. You may use **cin** for input, so assume no blanks. An example is below. A correct executable is available: **/home/Csc112-S-F16/strPerm.** A challenge will be to produce the permutations in the same order as the zyLab tests.

Submit your code to zyLab. Put all code in one file, so only **main.cpp** is needed. Declare the recursive function before main and put the function code after main.

For this problem, do your own work. You will likely have to research the problem as solution algorithms are out there, but feel free to come up with your own solution. You will find much information on the web about this.

You **MUST** document and comment your code in a neat, organized, and complete fashion. I.e., reading through your comments should explicitly describe how your algorithm works – even if this seems somewhat redundant with your code. Assume the reader is not a c++ expert. You grade for this problem will depend much on this documentation. Be sure to include references in your header comments.

Example run – Note: there is a new line after the last permutation

Input string:
cat //this is the input and not part of the output for the zyLab
Permutations:
cat
cta
act
atc
tca
tac