**Homework 3**

**Time due: 9:00 PM Tuesday, May 8**

1. Some online mapping software has the capability of presenting street maps with certain landmarks (e.g., hotels, restaurants, etc.) displayed. All landmarks have a name. All types of landmarks have a distinctive icon. Most types of landmarks are displayed with a yellow colored icon, but a few are some other color.

Declare and implement the classes named in the sample program below in such a way that the program compiles, executes, and produces the output shown. (Real mapping software would have snazzy graphics, but for now we'll stick to simple text output.) You must not change the implementations of display or main.

#include <iostream>

#include <string>

using namespace std;

*Your declarations and implementations would go here*

void display(const Landmark\* lm)

{

cout << "Display a " << lm->color() << " " << lm->icon() << " icon for "

<< lm->name() << "." << endl;

}

int main()

{

Landmark\* landmarks[4];

landmarks[0] = new Hotel("Westwood Rest Good");

// Restaurants have a name and seating capacity. Restaurants with a

// capacity under 40 have a small knife/fork icon; those with a capacity

// 40 or over have a large knife/fork icon.

landmarks[1] = new Restaurant("Bruin Bite", 30);

landmarks[2] = new Restaurant("La Morsure de l'Ours", 100);

landmarks[3] = new Hospital("UCLA Medical Center");

cout << "Here are the landmarks." << endl;

for (int k = 0; k < 4; k++)

display(landmarks[k]);

// Clean up the landmarks before exiting

cout << "Cleaning up." << endl;

for (int k = 0; k < 4; k++)

delete landmarks[k];

}

Output produced:

Here are the landmarks.

Display a yellow bed icon for Westwood Rest Good.

Display a yellow small knife/fork icon for Bruin Bite.

Display a yellow large knife/fork icon for La Morsure de l'Ours.

Display a blue H icon for UCLA Medical Center.

Cleaning up.

Destroying the hotel Westwood Rest Good.

Destroying the restaurant Bruin Bite.

Destroying the restaurant La Morsure de l'Ours.

Destroying the hospital UCLA Medical Center.

Decide which function(s) should be pure virtual, which should be non-pure virtual, and which could be non-virtual. Experiment to see what output is produced if you mistakenly make a function non-virtual when it should be virtual instead.

To force you to explore the issues we want you to, we'll put some constraints on your solution:

* + The Landmark class must not have a default constructor. The only constructor you may declare for Landmark must have one useful parameter.
  + Although the expression new Restaurant("Bruin Bite",30) is fine, the expression new Landmark("La Picadura del Oso") must produce a compilation error. (A client can create a particular *kind* of landmark, like a Restaurant, but is not allowed to create an object that is just a plain Landmark.)
  + Other than constructors and destructors (which can't be const), all member functions must be const member functions.
  + No two functions with non-empty bodies may have the same implementation. For example, there's a better way to deal with the name() function than to have each kind of landmark declare and identically implement a name function. (Notice that { return "shopping cart"; } and { return "purse"; } are different implementations.)
  + All data members must be declared private. You may declare member functions public or private. Your solution must *not* declare any protected members.

In a real program, you'd probably have separate Landmark.h, Landmark.cpp, Hotel.h, Hotel.cpp, etc., files. For simplicity for this problem, you may want to just put everything in one file. What you'll turn in for this problem will be a file named landmark.cpp containing the definitions and implementations of the four classes, and nothing more. (In other words, turn in only the program text that replaces *Your declarations and implementations would go here*.

1. Here are five functions, with descriptions of what they are supposed to do. They are incorrectly implemented. The first four take an array of strings and the number of strings to examine in the array; the last takes two arrays of strings and the number of strings to examine in each. (Note: If s is a string, s.empty() returns true if s is empty, and false otherwise. Also, for two strings s1 and s2, we will consider s1 to be less than s2 if s1 < s2 yields true.)
2. // Return true if any of the strings in the array is empty, false
3. // otherwise.
4. bool anyEmpty(const string a[], int n)
5. {
6. return false; // This is not always correct.
7. }
8. // Return the number of empty strings in the array.
9. int countEmpties(const string a[], int n)
10. {
11. return -999; // This is incorrect.
12. }
13. // Return the subscript of the first empty string in the array.
14. // If no element is empty, return -1.
15. int firstEmpty(const string a[], int n)
16. {
17. return -999; // This is incorrect.
18. }
19. // Return the subscript of the least string in the array (i.e.,
20. // the smallest subscript m such that there is no k for which
21. // a[k] < a[m]. If the array has no elements to examine, return -1.
22. int indexOfLeast(const string a[], int n)
23. {
24. return -999; // This is incorrect.
25. }
26. // If all n2 elements of a2 appear in the n1 element array a1, in
27. // the same order (though not necessarily consecutively), then
28. // return true; otherwise (i.e., if the array a1 does not include
29. // a2 as a not-necessarily-contiguous subsequence), return false.
30. // (Of course, if a2 is empty (i.e., n2 is 0), return true.)
31. // For example, if a1 is the 7 element array
32. // "stan" "kyle" "cartman" "kenny" "kyle" "cartman" "butters"
33. // then the function should return true if a2 is
34. // "kyle" "kenny" "butters"
35. // or
36. // "kyle" "cartman" "cartman"
37. // and it should return false if a2 is
38. // "kyle" "butters" "kenny"
39. // or
40. // "stan" "kenny" "kenny"
41. bool includes(const string a1[], int n1, const string a2[], int n2)
42. {
43. return false; // This is not always correct.
44. }

Replace the incorrect implementations of these functions with correct ones that use recursion in a useful way; your solution must not use the keywords while, for, or goto. You must not use global variables or variables declared with the keyword static, and you must not modify the function parameter lists. You may find it helpful to remember that a function parameter x declared T x[] for any type T means exactly the same thing as if it had been declared T\* x. If any of the parameters n, n1, or n2 is negative, act as if it were zero.

You will not receive full credit if for nonnegative n, anyEmpty, countEmpties, or firstEmpty causes string's empty() function to be called more than n times, or if indexOfLeast or includes causes a string comparison function (e.g. <= or ==) to be called more than n or n1 times, respectively. (For example, in determining how many of the n elements of an array are empty strings, you must not perform more than n tests to see whether a string is empty; you must solve this by causing each of the strings in the array to be tested no more than once.)

Each of these functions can be implemented in a way that meets the spec without calling any of the other four functions. (If you implement a function so that it *does* call one of the other functions, then it will probably not meet the limit stated in the previous paragraph.)

For this part of the homework, you will turn in one file named linear.cpp that contains the five functions and nothing more.

1. Replace the implementation of pathExists from [Homework 2](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/2/spec.html) with one that does not use an auxiliary data structure like a stack or queue, but instead uses recursion in a useful way. Here is pseudocode for a solution:
2. *If the start location is equal to the ending location, then we've*
3. *solved the maze, so return true.*
4. *Mark the start location as visted.*
5. *For each of the four directions,*
6. *If the location one step in that direction (from the start*
7. *location) is unvisited,*
8. *then call pathExists starting from that location (and*
9. *ending at the same ending location as in the*
10. *current call).*
11. *If that returned true,*
12. *then return true.*
13. *Return false.*

You may make the same simplifying assumptions that we allowed you to make for Homework 2 (e.g., that the maze contains only Xs and dots).

For this part of the homework, you will turn in one file named maze.cpp that contains the Coord class (if you use it) and the pathExists function and nothing more.

1. Replace the incorrect implementations of the countIncludes and the order functions below with correct ones that use recursion in a useful way. Except in the code for the split function that we give you below, your solution must not use the keywords while, for, or goto. You must not use any static or global variables, and you must not modify the function parameter lists. If any of the parameters n1, n2, or n is negative, act as if it were zero.
2. // Return the number of ways that all n2 elements of a2 appear
3. // in the n1 element array a1 in the same order (though not
4. // necessarily consecutively). The empty sequence appears in a
5. // sequence of length n1 in 1 way, even if n1 is 0.
6. // For example, if a1 is the 7 element array
7. // "stan" "kyle" "cartman" "kenny" "kyle" "cartman" "butters"
8. // then for this value of a2 the function must return
9. // "stan" "kenny" "cartman" 1
10. // "stan" "cartman" "butters" 2
11. // "kenny" "stan" "cartman" 0
12. // "kyle" "cartman" "butters" 3
13. int countIncludes(const string a1[], int n1, const string a2[], int n2)
14. {
15. return -999; // This is incorrect.
16. }
17. // Exchange two strings
18. void exchange(string& x, string& y)
19. {
20. string t = x;
21. x = y;
22. y = t;
23. }
24. // Rearrange the elements of the array so that all the elements
25. // whose value is < splitter come before all the other elements,
26. // and all the elements whose value is > splitter come after all
27. // the other elements. Upon return, firstNotLess is set to the index
28. // of the first element in the rearranged array that is >= splitter,
29. // or n if there is no such element, and firstGreater is set to
30. // the index of the first element that is > splitter, or n if
31. // there is no such element.
32. // In other words, upon return from the function, the array is a
33. // permutation of its original value such that
34. // \* for 0 <= i < firstNotLess, a[i] < splitter
35. // \* for firstNotLess <= i < firstGreater, a[i] == splitter
36. // \* for firstGreater <= i < n, a[i] > splitter
37. // All the elements < splitter end up in no particular order.
38. // All the elements > splitter end up in no particular order.
39. void split(string a[], int n, string splitter,
40. int& firstNotLess, int& firstGreater)
41. {
42. if (n < 0)
43. n = 0;
45. // It will always be the case that just before evaluating the loop
46. // condition:
47. // firstNotLess <= firstUnknown and firstUnknown <= firstGreater
48. // Every element earlier than position firstNotLess is < splitter
49. // Every element from position firstNotLess to firstUnknown-1 is
50. // == splitter
51. // Every element from firstUnknown to firstGreater-1 is not
52. // known yet
53. // Every element at position firstGreater or later is > splitter
55. firstNotLess = 0;
56. firstGreater = n;
57. int firstUnknown = 0;
58. while (firstUnknown < firstGreater)
59. {
60. if (a[firstUnknown] > splitter)
61. {
62. firstGreater--;
63. exchange(a[firstUnknown], a[firstGreater]);
64. }
65. else
66. {
67. if (a[firstUnknown] < splitter)
68. {
69. exchange(a[firstNotLess], a[firstUnknown]);
70. firstNotLess++;
71. }
72. firstUnknown++;
73. }
74. }
75. }
76. // Rearrange the elements of the array so that
77. // a[0] <= a[1] <= a[2] <= ... <= a[n-2] <= a[n-1]
78. // If n <= 1, do nothing.
79. void order(string a[], int n)
80. {
81. return; // This is not always correct.
82. }

(Hint: Using the split function, the order function can be written in fewer than eight short lines of code.)

You will not receive full credit if for nonnegative n1 and n2, the countIncludes function causes a string comparison function (e.g. <= or ==) to be called more than factorial(n1+1) / (factorial(n2)\*factorial(n1+1-n2)) times. The countIncludes function can be implemented in a way that meets the spec without calling any of the functions in problem 1. (If you implement it so that it *does* call one of those functions, then it will probably not meet the limit stated in the first sentence of this paragraph.)

For this part of the homework, you will turn in one file named tree.cpp that contains the four functions above and nothing more.

**Turn it in**

By Monday, May 7, there will be a link on the class webpage that will enable you to turn in this homework. Turn in one zip file that contains your solutions to the homework problems. The zip file must contain one to four of the four files landmark.cpp, linear.cpp, maze.cpp, and tree.cpp, depending on how many of the problems you solved. Your code must be such that if we insert it into a suitable test framework, it compiles. (In other words, it must have no missing semicolons, extra parentheses, undeclared variables, and so on. Our test framework will have the necessary #includes, etc.)