Winter 2017 CS 32

**Homework 3**

**Time due: 9:00 PM ~~Tuesday, February 14~~ Wednesday, February 15**

1. You are developing software for Crapple's next device, the iBad. Among its capabilities is the ability to access video, audio, and text files. Every file has a name. Each kind of file has a distinctive action that's taken when it is opened. A client can ask to redisplay a file; for most kinds of files, this just refreshes the screen, but some kinds of files may do something interesting.

Declare and implement the classes named in the sample program below in such a way that the program compiles, executes, and produces exactly the output shown. (A real implementation would actually play videos and show pictures, but for now we'll stick to simple text output.) Do not change the implementations of openAndRedisplay or main.

#include <iostream>

#include <string>

using namespace std;

*Your declarations and implementations would go here*

void openAndRedisplay(const File\* f)

{

cout << f->name() << ": ";

f->open();

cout << endl << "Redisplay: ";

f->redisplay();

cout << endl;

}

int main()

{

File\* files[4];

files[0] = new TextMsg("fromFred.txt");

// Videos have a name and running time

files[1] = new Video("goblin.mpg", 3780);

files[2] = new Picture("kitten.jpg");

files[3] = new Picture("baby.jpg");

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

openAndRedisplay(files[k]);

// Clean up the files before exiting

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

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

delete files[k];

}

Output produced:

fromFred.txt: open text message

Redisplay: refresh the screen

goblin.mpg: play 3780 second video

Redisplay: replay video

kitten.jpg: show picture

Redisplay: refresh the screen

baby.jpg: show picture

Redisplay: refresh the screen

Cleaning up.

Destroying fromFred.txt, a text message

Destroying goblin.mpg, a video

Destroying the picture kitten.jpg

Destroying the picture baby.jpg

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:

* + You must not declare any struct or class other than File, TextMsg, Video, and Picture.
  + The File class must not have a default constructor. The only constructor you may declare for File must have exactly one parameter. That parameter must be of a builtin type or of type string, and it must be a useful parameter.
  + Although the expression new Video("scarletheart.mpg",3540) is fine, the expression new File("readme") and new File(0) must produce compilation errors. (A client can create a particular *kind* of file, like a Video, but is not allowed to create an object that is just a plain File.)
  + 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 implementations that have the same effect for a caller. For example, there's a better way to deal with the name()function than to have each kind of file  declare and identically implement a name function. (Notice that { cout << "show picture"; } and { cout << "open text message"; } do not have the same effect, but { cout << "show picture"; } and { cout << "show " << "picture"; } have the same effect, which is to write show picture to cout.
  + No implementation of a name() function may call any other function.
  + All data members must be declared private. You may declare member functions public or private. Your solution must *not* declare any protected members (which we're not covering in this class).

In a real program, you'd probably have separate File.h, File.cpp, TextMsg.h, TextMsg.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 file.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. The following is a declaration of a function that takes a double and returns true if a particular property of that double is true, and false otherwise. (Such a function is called a *predicate*.)
2. bool somePredicate(double x);

Here is an example of an implementation of the predicate *x is negative*:

bool somePredicate(double x)

{

return x < 0;

}

Here is an example of an implementation of the predicate *sin ex is greater than cos x*:

bool somePredicate(double x)

{

return sin(exp(x)) > cos(x); // include <cmath> for std::sin, etc.

}

Here are five functions, with descriptions of what they are supposed to do. They are incorrectly implemented. The first four take an array of doubles and the number of doubles to examine in the array; the last takes two arrays of doubles and the number of doubles to examine in each:

// Return false if the somePredicate function returns false for at

// least one of the array elements; return true otherwise.

bool allTrue(const double a[], int n)

{

return false; // This is not always correct.

}

// Return the number of elements in the array for which the

// somePredicate function returns false.

int countFalse(const double a[], int n)

{

return -999; // This is incorrect.

}

// Return the subscript of the first element in the array for which

// the somePredicate function returns false. If there is no such

// element, return -1.

int firstFalse(const double a[], int n)

{

return -999; // This is incorrect.

}

// Return the subscript of the smallest double in the array (i.e.,

// the one whose value is <= the value of all elements). If more

// than one element has the same smallest value, return the smallest

// subscript of such an element. If the array has no elements to

// examine, return -1.

int indexOfMin(const double a[], int n)

{

return -999; // This is incorrect.

}

// If all n2 elements of a2 appear in the n1 element array a1, in

// the same order (though not necessarily consecutively), then

// return true; otherwise (i.e., if the array a1 does not include

// a2 as a not-necessarily-contiguous subsequence), return false.

// (Of course, if a2 is empty (i.e., n2 is 0), return true.)

// For example, if a1 is the 7 element array

// 10 50 40 20 50 40 30

// then the function should return true if a2 is

// 50 20 30

// or

// 50 40 40

// and it should return false if a2 is

// 50 30 20

// or

// 10 20 20

bool includes(const double a1[], int n1, const double a2[], int n2)

{

return false; // This is not always correct.

}

Your implementations of those first three functions must call the function named somePredicate where appropriate instead of hardcoding a particular expression like x < 0 or sin(exp(x)) > cos(x). (When you test your code, we don't care what predicate you have the function named somePredicate implement: *x < 0* or *x == 42* or *sqrt(log(x\*x+1)) > 2* or whatever, is fine.)

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 must not use any references or pointers as parameters except for the parameters representing arrays. (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.

Here is an example of an implementation of allTrue that does *not* satisfy these requirements because it doesn't use recursion and it uses the keyword for:

bool allTrue(const double a[], int n)

{

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

{

if ( ! somePredicate(a[k]))

return false;

}

return true;

}

You will not receive full credit if the allTrue, countFalse, or firstFalse functions cause each value somePredicate returns to be examined more than once. Consider all operations that a function performs that compares two doubles (e.g. <=, ==, etc.). You will not receive full credit if for nonnegative n, the indexOfMin function causes operations like these to be performed more than n times, or the includes function causes them to be performed more than n1 times. For example, this non-recursive (and thus unacceptable for this problem) implementation of indexOfMin performs a <= comparison of two doubles many, many more than n times, which is also unacceptable:

int indexOfMin(const double a[], int n)

{

for (int k1 = 0; k1 < n; k1++)

{

int k2;

for (k2 = 0; k2 < n && a[k1] <= a[k2]; k2++)

;

if (k2 == n)

return k1;

}

return -1;

}

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: no implementation of somePredicate and no main routine. (Our test framework will precede the functions with an implementation of a function named somePredicate that takes a double and returns a bool.)

1. Replace the implementation of pathExists from [Homework 2](http://web.cs.ucla.edu/classes/winter17/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.*

(If you wish, you can implement the pseudocode for loop with a series of four if statements instead of a loop.)

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 separate function that we give you below, 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 must not use any references or pointers as parameters except for the parameters representing arrays and the parameters of the exchange function we provided. 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. // 10 50 40 20 50 40 30
8. // then for this value of a2 the function must return
9. // 10 20 40 1
10. // 10 40 30 2
11. // 20 10 40 0
12. // 50 40 30 3
13. int countIncludes(const double a1[], int n1, const double a2[], int n2)
14. {
15. return -999; // This is incorrect.
16. }
17. // Exchange two doubles
18. void exchange(double& x, double& y)
19. {
20. double 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 > separator come before all the other elements,
26. // and all the elements whose value is < separator come after all
27. // the other elements. Upon return, firstNotGreater is set to the
28. // index of the first element in the rearranged array that is
29. // <= separator, or n if there is no such element, and firstLess is
30. // set to the index of the first element that is < separator, or n
31. // if 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 < firstNotGreater, a[i] > separator
35. // \* for firstNotGreater <= i < firstLess, a[i] == separator
36. // \* for firstLess <= i < n, a[i] < separator
37. // All the elements > separator end up in no particular order.
38. // All the elements < separator end up in no particular order.
39. void separate(double a[], int n, double separator,
40. int& firstNotGreater, int& firstLess)
41. {
42. if (n < 0)
43. n = 0;
45. // It will always be the case that just before evaluating the loop
46. // condition:
47. // firstNotGreater <= firstUnknown and firstUnknown <= firstLess
48. // Every element earlier than position firstNotGreater is > separator
49. // Every element from position firstNotGreater to firstUnknown-1 is
50. // == separator
51. // Every element from firstUnknown to firstLess-1 is not known yet
52. // Every element at position firstLess or later is < separator
54. firstNotGreater = 0;
55. firstLess = n;
56. int firstUnknown = 0;
57. while (firstUnknown < firstLess)
58. {
59. if (a[firstUnknown] < separator)
60. {
61. firstLess--;
62. exchange(a[firstUnknown], a[firstLess]);
63. }
64. else
65. {
66. if (a[firstUnknown] > separator)
67. {
68. exchange(a[firstNotGreater], a[firstUnknown]);
69. firstNotGreater++;
70. }
71. firstUnknown++;
72. }
73. }
74. }
75. // Rearrange the elements of the array so that
76. // a[0] >= a[1] >= a[2] >= ... >= a[n-2] >= a[n-1]
77. // If n <= 1, do nothing.
78. void order(double a[], int n)
79. {
80. return; // This is not always correct.
81. }

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

Consider all operations that a function performs that compares two doubles (e.g. <=, ==, etc.). You will not receive full credit if for nonnegative n1 and n2, the countIncludesfunction causes operations like these 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 2. (If you implement it so that it *does* call one of those functions, then it will probably not meet the limit stated in 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, February 13, 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 file.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 a suitable test framework with a main routine and appropriate #include directives, it compiles. (In other words, it must have no missing semicolons, unbalanced parentheses, undeclared variables, etc.) You must verify that your code works correctly using g32 and either Visual C++ or clang++.