Homework 3

Time due: 11:00 PM Tuesday, May 9

1. You are developing a computer game for pre-schoolers that features different kinds of domesticated animals (e.g., ducks and cats). Every animal has a name (e.g., Daffy or Fluffy). Every kind of animal makes a distinctive sound when it speaks (e.g. "Quack" or "Meow"). When most kinds of animals move, they walk, but a few kinds may do something different.

   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. (The real game will have all sorts of snazzy graphics and audio, but for now we'll stick to simple text output.) You must not change the implementations of `animate` or `main`.

```
#include <iostream>
#include <string>
using namespace std;

Your declarations and implementations would go here

void animate(const Animal* a)
{
    a->speak();
    cout << "!  My name is " << a->name()
         << ".  Watch me " << a->moveAction() << "!\n";
}

int main()
{
    Animal* animals[4];
    animals[0] = new Cat("Fluffy");
      // Pigs have a name and a weight in pounds.  Pigs under 170
      // pounds oink; pigs weighing at least 170 pounds grunt.
    animals[1] = new Pig("Napoleon", 190);
    animals[2] = new Pig("Wilbur", 50);
    animals[3] = new Duck("Daffy");

    cout << "Here are the animals." << endl;
    for (int k = 0; k < 4; k++)
        animate(animals[k]);

      // Clean up the animals before exiting
    cout << "Cleaning up." << endl;
    for (int k = 0; k < 4; k++)
        delete animals[k];
```

```
}
```

Output produced:

```
Here are the animals.
Meow!  My name is Fluffy.  Watch me walk!
Grunt!  My name is Napoleon.  Watch me walk!
Oink!  My name is Wilbur.  Watch me walk!
Quack!  My name is Daffy.  Watch me swim!
Cleaning up.
Destroying Fluffy the cat
Destroying Napoleon the pig
Destroying Wilbur the pig
Destroying Daffy the duck
```

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 Animal, Cat, Pig, and Duck.
- The Animal class must not have a default constructor. The only constructor you may declare for Animal 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 Pig("Snowball", 170)` is fine, the expressions `new Animal("Midnight")` and `new Animal(0)` must produce compilation errors. (A client can create a particular *kind* of animal object, like a `Pig` object, but is not allowed to create an object that is just a plain `Animal`.)
- 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 animal declare and identically implement a name function. (Notice that `{ return "walk"; }` and `{ string s("wa"); return s + "lk"; }` have the

same effect. And notice that `{ cout << "Moo"; }` and `{ cout << "Squeak"; }` do not have the same effect.)

- No implementation of a `name()` function may call any other function.
- No class may have a data member whose value is identical for every object of a particular class type.
- 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 Animal.h, Animal.cpp, Cat.h, Cat.cpp, etc., files. For simplicity for this problem, you'll want to just put everything in one file. What you'll turn in for this problem will be a file named `animal.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*.)

2. The following is a declaration of a function that takes a string and returns true if a particular property of that string is true, and false otherwise. (Such a function is called a *predicate*.)
3. `bool somePredicate(string s);`

Here is an example of an implementation of the predicate *s is empty*:

```
bool somePredicate(string s)
{
    return s.empty();
}
```

Here is an example of an implementation of the predicate *s contains exactly 10 digits*:

```
bool somePredicate(string s)
{
    int nDigits = 0;
    for (int k = 0; k != s.size(); k++)
    {
        if (isdigit(s[k]))
            nDigits++;
    }
    return nDigits == 10;
}
```

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:

```
    // Return false if the somePredicate function returns false
for at
    // least one of the array elements; return true otherwise.
  bool allTrue(const string 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 string 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 string a[], int n)
  {
      return -999;  // This is incorrect.
  }

    // Return the subscript of the least string in the array
(i.e.,
    // the smallest subscript m such that a[m] <= a[k] for all
    // k from 0 to n-1).  If the array has no elements to examine,
    // return -1.
  int indexOfLeast(const string 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
    //    "stan" "kyle" "cartman" "kenny" "kyle" "cartman"
"butters"
    // then the function should return true if a2 is
    //    "kyle" "kenny" "butters"
    // or
    //    "kyle" "cartman" "cartman"
```

```
    // and it should return false if a2 is
    //    "kyle" "butters" "kenny"
    // or
    //    "stan" "kenny" "kenny"
  bool includes(const string a1[], int n1, const string 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 `a[k].empty()` or `a[k].size() == 42`. (When you test your code, we don't care what predicate you have the function named `somePredicate` implement: *s.empty()* or *s.size() == 42* 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 string 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 strings (e.g. `<=`, `==`, etc.). You will not receive full credit if for nonnegative `n`, the `indexOfLeast` 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 `indexOfLeast` performs a <= comparison of two strings many, many more than `n` times, which is also unacceptable:

```
int indexOfLeast(const string 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 #include directives, no implementation of `somePredicate` and no main routine. (Our test framework will precede the functions with appropriate #include directives and our own implementation of a function named `somePredicate` that takes a string and returns a bool.)

4. Replace the implementation of `pathExists` from [Homework 2] 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:

```
5.  If the start location is equal to the ending location, then we've
6.      solved the maze, so return true.
7.  Mark the start location as visted.
8.  For each of the four directions,
9.      If the location one step in that direction (from the start
10.                 location) is unvisited,
11.                     then call pathExists starting from that location (and
12.                                 ending at the same ending location as in the
13.                                 current call).
14.                          If that returned true,
15.                              then return true.
16.         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.

17. 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.

```
  // Return the number of ways that all n2 elements of a2 appear
  // in the n1 element array a1 in the same order (though not
  // necessarily consecutively).  The empty sequence appears in a
  // sequence of length n1 in 1 way, even if n1 is 0.
  // For example, if a1 is the 7 element array
  //    "stan" "kyle" "cartman" "kenny" "kyle" "cartman" "butters"
  // then for this value of a2            the function must return
  //    "stan" "kenny" "cartman"                      1
  //    "stan" "cartman" "butters"                    2
  //    "kenny" "stan" "cartman"                      0
  //    "kyle" "cartman" "butters"                    3
int countIncludes(const string a1[], int n1, const string a2[], int n2)
{
    return -999;  // This is incorrect.
}

  // Exchange two strings
void exchange(string& x, string& y)
{
    string t = x;
    x = y;
    y = t;
}

```

```
 // Rearrange the elements of the array so that all the elements
 // whose value is < separator come before all the other elements,
 // and all the elements whose value is > separator come after all
 // the other elements.  Upon return, firstNotLess is set to the
 // index of the first element in the rearranged array that is
 // >= separator, or n if there is no such element, and firstGreater is
 // set to the index of the first element that is > separator, or n
 // if there is no such element.
 // In other words, upon return from the function, the array is a
 // permutation of its original value such that
 //   * for 0 <= i < firstNotLess, a[i] < separator
 //   * for firstNotLess <= i < firstGreater, a[i] == separator
 //   * for firstGreater <= i < n, a[i] > separator
 // All the elements < separator end up in no particular order.
 // All the elements > separator end up in no particular order.
void separate(string a[], int n, string separator,
                                    int& firstNotLess, int& firstGreater)
{
    if (n < 0)
        n = 0;

      // It will always be the case that just before evaluating the loop
      // condition:
      //  firstNotLess <= firstUnknown and firstUnknown <= firstGreater
      //  Every element earlier than position firstNotLess is < separator
      //  Every element from position firstNotLess to firstUnknown-1 is
      //    == separator
      //  Every element from firstUnknown to firstGreater-1 is not known yet
      //  Every element at position firstGreater or later is > separator

    firstNotLess = 0;
    firstGreater = n;
    int firstUnknown = 0;
    while (firstUnknown < firstGreater)
    {
        if (a[firstUnknown] > separator)
        {
            firstGreater--;
            exchange(a[firstUnknown], a[firstGreater]);
```

```
            }
            else
            {
                if (a[firstUnknown] < separator)
                {
                    exchange(a[firstNotLess], a[firstUnknown]);
                    firstNotLess++;
                }
                firstUnknown++;
            }
        }
    }

      // Rearrange the elements of the array so that
      // a[0] <= a[1] <= a[2] <= ... <= a[n-2] <= a[n-1]
      // If n <= 1, do nothing.
    void order(string a[], int n)
    {
        return;  // This is not always correct.
    }
```

(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 strings (e.g. `<=`, `==`, etc.). You will not receive full credit if for nonnegative `n1` and `n2`, the `countIncludes`function 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, May 8, 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 `animal.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 with a main routine and appropriate #include directives, it compiles. (In other words, it must have no missing semicolons, unbalanced parentheses, undeclared variables, etc.)