Homework 3

Time due: 9:00 PM Tuesday, February 10

1. You are developing a computer game based on the 1969 parody, *Bored of the Rings*. The game has several types of characters (e.g., elves and dwarves). Every character has a name. Each type of character has a distinctive weapon of choice. When most kinds of characters attack, they rush toward the enemy, but a few 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.)

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

Your declarations and implementations would go here

void strike(const Character* cp)
{
    cout << cp->name() << ", wielding ";
    cp->printWeapon();
    cout << ", " << cp->attackAction() << "." << endl;
}

int main()
{
    Character* characters[4];
    characters[0] = new Dwarf("Gimlet");
        // Elves have a name and initial number of arrows in their quiver
    characters[1] = new Elf("Legolam", 10);
    characters[2] = new Boggie("Frito");
    characters[3] = new Boggie("Spam");

    cout << "The characters strike!" << endl;
    for (int k = 0; k < 4; k++)
        strike(characters[k]);

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

   Output produced:

```
The characters strike!
Gimlet, wielding an axe, rushes toward the enemy.
Legolam, wielding a bow and quiver of 10 arrows, rushes toward the enemy.
Frito, wielding a short sword, whimpers.
Spam, wielding a short sword, whimpers.
Cleaning up
```

```
Destroying Gimlet the dwarf
Destroying Legolam the elf
Destroying Frito the boggie
Destroying Spam the boggie
```

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 Character, Dwarf, Elf, and Boggie.
- The Character class must not have a default constructor. The only constructor you may declare for Character 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 Elf("Orlon", 8)` is fine, the expressions `new Character("Goodgulf")` and `new Character(0)` must produce compilation errors. (A client can create a particular *kind* of character object, like an `Elf` object, but is not allowed to create an object that is just a plain `Character`.)
- 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 character declare and identically implement a name function. (Notice that `{ return "rushes toward the enemy"; }` and `{ string s("rushes toward"); return s + " the enemy"; }` have the same effect. And notice that `{ cout << "a pointed stick"; }` and `{ cout << "banana"; }` do not have the same effect.)
- 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 Character.h, Character.cpp, Dwarf.h, Dwarf.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 `character.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 double and returns true if a particular property of that double is true, and false otherwise. (Such a function is called a *predicate*.)

```
        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* e^x *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 true if the somePredicate function returns true for at
  // least one of the array elements, false otherwise.
bool anyTrue(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 true.
int countTrue(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 true.  If there is no such
  // element, return -1.
int firstTrue(const double a[], int n)
{
    return -999;  // This is incorrect.
}

  // Return the subscript of the smallest element 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*, *x == 42*, *sqrt(log(x*x+1)) > 5*, 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 `anyTrue` that does *not* satisfy these requirements because it doesn't use recursion:

```
        bool anyTrue(const double a[], int n)
        {
            for (int k = 0; k < n; k++)
            {
                if (somePredicate(a[k]))
                    return true;
            }
            return false;
        }
```

You will not receive full credit if the `anyTrue`, `countTrue`, or `firstTrue` 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. (Our test framework will precede the functions with an implementation of a function named `somePredicate` that takes a double and returns a bool.)

3. 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:

```
If the start location is equal to the ending location, then we've
    solved the maze, so return true.
Mark the start location as visted.
For each of the four directions,
    If the location one step in that direction (from the start
        location) is unvisited,
            then call pathExists starting from that location (and
                        ending at the same ending location as in the
                        current call).
                 If that returned true,
                     then return true.
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.

4. 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
  //    10 50 40 20 50 40 30
  // then for this value of a2     the function must return
  //    10 20 40                        1
  //    10 40 30                        2
  //    20 10 40                        0
  //    50 40 30                        3
int countIncludes(const double a1[], int n1, const double a2[], int n2)
{
    return -999;  // This is incorrect.
}

  // Exchange two doubles
void exchange(double& x, double& y)
{
    double 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, firstNotGreater 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 firstLess 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 < firstNotGreater, a[i] > separator
  //   * for firstNotGreater <= i < firstLess, a[i] == separator
  //   * for firstLess <= 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(double a[], int n, double separator,
                                    int& firstNotGreater, int& firstLess)
{
    if (n < 0)
        n = 0;

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

    firstNotGreater = 0;
    firstLess = n;
    int firstUnknown = 0;
    while (firstUnknown < firstLess)
    {
        if (a[firstUnknown] < separator)
        {
            firstLess--;
            exchange(a[firstUnknown], a[firstLess]);
        }
        else
        {
            if (a[firstUnknown] > separator)
            {
                exchange(a[firstNotGreater], a[firstUnknown]);
                firstNotGreater++;
            }
            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(double 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 doubles (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, February 9, 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 `character.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.)