# Homework 3

**Time due: 11:00 PM Tuesday, February 11**

1. You are developing event management software for a prominent venue. Every event has a name. Every type of event has a significant need (e.g. hoops for a basketball game or a stage for a concert). Most types of events are sporting events, but a few are not.

   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. 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 Event* e)
{
    cout << e->name() << ": ";
    if (e->isSport())
        cout << "(sport) ";
    cout << "needs " << e->need() << endl;
}

int main()
{
    Event* events[4];
    events[0] = new BasketballGame("Lakers vs. Suns");
      // Concerts have a name and a genre.
    events[1] = new Concert("Banda MS", "banda");
    events[2] = new Concert("KISS", "hard rock");
    events[3] = new HockeyGame("Kings vs. Flames");

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

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

   Output produced:

```
Here are the events.
Lakers vs. Suns: (sport) needs hoops
Banda MS: needs a stage
KISS: needs a stage
```

```
Kings vs. Flames: (sport) needs ice
Cleaning up.
Destroying the Lakers vs. Suns basketball game
Destroying the Banda MS banda concert
Destroying the KISS hard rock concert
Destroying the Kings vs. Flames hockey game
```

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 Event, BasketballGame, Concert, and HockeyGame.
- The Event class must not have a default constructor. The only constructor you may declare for Event must have exactly one parameter. That parameter must be of type string, and it must be a useful parameter.
- Although the expression `new Concert("BLACKPINK", "K-pop")` is fine, the expression `new Event("Sparks vs. Aces")` must produce a compilation error. (A client can create a particular *kind* of event object, like a `Concert`, but is not allowed to create an object that is just a plain `Event`.)
- 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, or 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 event declare and identically implement a name function. (For people looking for loopholes, notice that `{ return "hoops"; }` and `{ return "ice"; }` do not have the same effect, but for, say, a function returning a double, `{ return 1.0; }` and `{ int i = 4; return i-3; }` *do* have the same effect, since in the second case, the int 1 that is computed will be converted to the double 1.0 to be returned.)
- 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). Your solution must not contain the word `friend`.

In a real program, you'd probably have separate Event.h, Event.cpp, Basketball.h, Basketball.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 `event.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 false for at
  // least one of the array elements; return false otherwise.
bool anyFalse(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.,
  // return the smallest subscript m such that a[m] <= a[k] for all
  // k from 0 to n-1).  If the function is told to examine no
  // elements, return -1.
int positionOfSmallest(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 contain
  // 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 contains(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)) > 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 `anyFalse` that does *not* satisfy these requirements because it doesn't use recursion and it uses the keyword `for`:

```
bool anyFalse(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 `anyFalse`, `countTrue`, or `firstTrue` functions causes the value that each call of `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 `positionOfSmallest` function causes operations like these to be performed more than `n` times, or the `contains` function causes them to be performed more than `n1` times. For example, this non-recursive (and thus unacceptable for this problem) implementation of `positionOfSmallest` performs a <= comparison of two doubles many, many more than `n` times, which is also unacceptable:

```
int positionOfSmallest(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 #include directives, no using namespace std;, no implementation of `somePredicate` and no main routine. (Our test framework will precede the functions with appropriate #include directives, using statement, and our own 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 `countContains` 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` and `separate` functions 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 countContains(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 `countContains` function causes operations like these to be called more than `factorial(n1+1) / (factorial(n2)*factorial(n1+1-n2))` times. The `countContains` 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 10, 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 `event.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 and using statements, it compiles. (In other words, it must have no missing semicolons, unbalanced parentheses, undeclared variables, etc.)