Project 2: Hope Springs Eternal, So Does Chip! (due Friday, July 20th at 11:59 PM)

Every year, there seems to be annual ritual. In the world, we've been told there are two things guaranteed in life: death and taxes. However, someone forgot to mention that this applies to another thing: changing coaching staffs in any sport. At UCLA, this just happened in football with the firing of head coach Jim Mora and the hiring of head coach Chip Kelly. But, it's never just about the head coach. It's also the assistant coaches (and their families) that get affected as well. Some get to stay (cue up "Elian can stay!"), while some have to go (cue up either "Bye, Felicia!" or as our President said on a TV show called The Apprentice, "You're fired!"). Just food for thought.

In this project, you will write the implementation of the CoachingStaff using a doubly linked list, which should be sorted alphabetically according to last name, then first name. You will also implement a couple of algorithms that operate on a CoachingStaff.

Implement CoachingStaff

Consider the following CoachingStaff interface:

```
typedef std::string IType;

class CoachingStaff
{
  public:
    CoachingStaff();       // Create an empty CoachingStaff list

    bool noCoaches() const;  // Return true if the CoachingStaff list
                             // is empty, otherwise false.

    int numberOfCoaches() const;  // Return the number of elements in
                                  // the CoachingStaff list.

    bool hireCoach(const std::string& firstName, const std::string&
                lastName, const IType& value);
      // If the full name (both the first and last name) is not equal
      // to any full name currently in the list then add it and return
      // true. Elements should be added according to their last name.
      // Elements with the same last name should be added according to
      // their first names. Otherwise, make no change to the list and
      // return false (indicating that the name is already in the
      // list).

    bool renameCoach(const std::string& firstName, const std::string&
lastName, const IType& value);
      // If the full name is equal to a full name currently in the
      // list, then make that full name no longer map to the value it
      // currently maps to, but instead map to the value of the third
```

```
      // parameter; return true in this case. Otherwise, make no
      // change to the list and return false.

    bool hireOrRename(const std::string& firstName, const std::string&
lastName, const IType& value);
      // If full name is equal to a name currently in the list, then
      // make that full name no longer map to the value it currently
      // maps to, but instead map to the value of the third parameter;
      // return true in this case. If the full name is not equal to
      // any full name currently in the list then add it and return
      // true. In fact, this function always returns true.

    bool fireCoach(const std::string& firstName, cont std::string&
lastName);
      // If the full name is equal to a full name currently in the
      // list, remove the full name and value from the list and return
      // true.  Otherwise, make no change to the list and return
      // false.

    bool coachOnStaff(const std::string& firstName, const std::string&
lastName) const;
      // Return true if the full name is equal to a full name
      // currently in the list, otherwise false.

    bool findCoach(const std::string& firstName, const std::string&
lastName, IType& value) const;
      // If the full name is equal to a full name currently in the
      // list, set value to the value in the list that that full name
      // maps to, and return true.  Otherwise, make no change to the
      // value parameter of this function and return false.

    bool whichCoach(int i, std::string& firstName, std::string&
lastName, IType& value) const;
      // If 0 <= i < size(), copy into firstName, lastName and value
      // parameters the corresponding information of the element at
      // position i in the list and return true.  Otherwise, leave the
      // parameters unchanged and return false. (See below for details
      // about this function.)

    void changeStaff(CoachingStaff& other);
      // Exchange the contents of this list with the other one.
};
```

The `hireCoach` function primarily places elements so that they are sorted in the list based on last name. If there are multiple entries with the same last name then those elements, with the same last name, are added so that they are sorted by their first name. In other words, this code fragment

```
CoachingStaff o;

o.hireCoach ("Chip", "Kelly", 54);
o.hireCoach ("Dana", "Bible", 64);
o.hireCoach ("Jimmie", "Dougherty", 39);
o.hireCoach ("DeShaun", "Foster", 38);
o.hireCoach ("Derek", "Sage", 40);
o.hireCoach ("Justin", "Frye", 34);

for (int n = 0; n < o.numberOfCoaches(); n++)
{
    string first;
    string last;
    int val;
    o.whichCoach (n, first, last, val);
    cout << first << " " << last << " " << val << endl;
}
```

must result in the output:

```
Dana Bible 64
Jimmie Dougherty 39
DeShaun Foster 38
Justin Frye 34
Chip Kelly 54
Derek Sage 40
```

works

Notice that the empty string is just as good a string as any other; you should not treat it in any special way:

```
CoachingStaff fortyTimes;

fortyTimes.hireCoach("Jerry", "Azzinaro", 6.99);
assert(!fortyTimes.coachOnStaff ("",""));
fortyTimes.hireCoach("Vince", "Oghobaase", 5.19);
fortyTimes.hireCoach("", "", 4.00);
fortyTimes.hireCoach("Roy", "Manning", 4.7);
assert(fortyTimes.coachOnStaff ("", ""));
fortyTimes.fireCoach("Vince", "Oghobaase");
assert(fortyTimes.numberOfCoaches() == 3
       && fortyTimes.coachOnStaff("Jerry", "Azzinaro")
       && fortyTimes.coachOnStaff ("Roy", "Manning")
       && fortyTimes.coachOnStaff ("", ""));
```

works

When comparing keys for `hireCoach`, `renameCoach`, `hireOrRename`, `fireCoach`, `coachOnStaff`, and `findCoach`, just use the == or != operators provided for the string type by the library. These do case-sensitive comparisons, and that's fine.

For this project, implement this `CoachingStaff` interface using a doubly-linked list. (You must not use any container from the C++ library.)

For your implementation, if you let the compiler write the destructor, copy constructor, and assignment operator, they will do the wrong thing, so you will have to declare and implement these public member functions as well:

Destructor

When a `CoachingStaff` is destroyed, all dynamic memory must be deallocated.

Copy Constructor

When a brand new `CoachingStaff` is created as a copy of an existing `CoachingStaff`, a deep copy should be made.

Assignment Operator

When an existing `CoachingStaff` (the left-hand side) is assigned the value of another `CoachingStaff` (the right-hand side), the result must be that the left-hand side object is a duplicate of the right-hand side object, with no memory leak (i.e. no memory from the old value of the left-hand side should be still allocated yet inaccessible).

Notice that there is no a priori limit on the maximum number of elements in the `CoachingStaff` (so `addOrChange` should always return true). Also, if a `CoachingStaff` has a size of n, then the values of the first parameter to the get member function are 0, 1, 2, ..., n – 1; for other values, it returns false without setting its parameters.

Implement Some Non-Member Functions

Using only the *public* interface of `CoachingStaff`, implement the following two functions. (Notice that they are non-member functions; they are not members of `CoachingStaff` or any other class.)

```
bool mergeStaffs(const CoachingStaff & csOne,
                 const CoachingStaff & csTwo,
                 CoachingStaff & csMerged);
```

When this function returns, `csMerged` must consist of pairs determined by these rules:

If a full name appears in exactly one of `csOne` and `csTwo`, then `csMerged` must contain an element consisting of that full name and its corresponding value.

If a full name appears in both `csOne` and `csTwo`, with the same corresponding value in both, then `csMerged` must contain an element with that full name and value.

When this function returns, `csMerged` must contain no elements other than those required by these rules. (You must not assume `csMerged` is empty when it is passed in to this function; it might not be.)

If there exists a full name that appears in both `csOne` and `csTwo`, but with different corresponding values, then this function returns false; if there is no full name like this, the function returns true. Even if the function returns false, result must be constituted as defined by the above rules.

For example, suppose a `CoachingStaff` maps the full name to integers. If `csOne` consists of these three elements

```
"Bill" "Yoast" 456   "Herb" "Tyrell" 123   "Herman" "Boone" 789
```

and `csTwo` consists of

```
"Herman" "Boone" 789   "Doc" "Hines" 321
```

then no matter what value it had before, `csMerged` must end up as a list consisting of

```
"Herman" "Boone" 789   "Doc" "Hines" 321   "Herb" "Tyrell" 123
"Bill" "Yoast" 456
```

and `mergeStaffs` must return true.

If instead, `csOne` consists of

```
"Bill" "Yoast" 456   "Herb" "Tyrell" 123   "Herman" "Boone" 789
```

and `csTwo` consists of

```
"Herman" "Boone" 654   "Doc" "Hines" 321
```

then no matter what value it had before, `csMerged` must end up as a list consisting of

```
"Doc" "Hines" 321   "Herb" "Tyrell" 123   "Bill" "Yoast" 456
```

and `mergeStaffs` must return false.

```
void searchStaff (const std::string& fsearch,
                  const std::string& lsearch,
                  const CoachingStaff& csOne,
                  CoachingStaff& csResult);
```

When this function returns, `csResult` must contain a copy of all the elements in `csOne` that match the search terms; it must not contain any other elements. You can wildcard the first name, last name or both by supplying "*". (You must not assume result is empty when it is passed in to this function; it may not be.)

For example, if `mbb` consists of the three elements

```
"Steve" "Alford" 53   "Tyus" "Edney" 45   "Kory" "Alford" 26
```

and the following call is made:

```
searchStaff("*", "Alford", mbb, result);
```

then no matter what value it had before, `csResult` must end up as a `CoachingStaff` consisting of

```
"Kory" "Alford" 26   "Steve" "Alford" 53
```

If instead, `wbb` were

```
"Cori" "Close" 46   "Shannon" "Perry" 42   "Cori" "Doe" 27
```

and the following call is made:

```
searchStaff("Cori", "*", wbb, result);
```

then no matter what value it had before, result must end up as a list consisting of

```
"Cori" "Close" 46   "Cori" "Doe" 27
```

If the following call is made:

```
searchStaff("*", "*", mbb, result);
```

then no matter what value it had before, `result` must end up being a copy of `mbb`.

Be sure these functions behave correctly in the face of aliasing: What if `wbb` and `result` refer to the same `CoachingStaff`, for example?

Other Requirements

Regardless of how much work you put into the assignment, your program will receive a low score for correctness if you violate these requirements:

- Your class definition, declarations for the two required non-member functions, and the implementations of any functions you choose to inline must be in a file named `CoachingStaff.h`, which must have appropriate include guards. The implementations of the functions you declared in `CoachingStaff.h` that you did not inline must be in a file named `CoachingStaff.cpp`. Neither of those files may have a main routine (unless it's commented out). You may use a

separate file for the main routine to test your `CoachingStaff` class; you won't turn in that separate file.

- Except to add a destructor, copy constructor, assignment operator, and dump function (described below), you must not add functions to, delete functions from, or change the public interface of the `CoachingStaff` class. You must not declare any additional struct/class outside the `CoachingStaff` class, and you must not declare any public struct/class inside the `CoachingStaff` class. You may add whatever private data members and private member functions you like, and you may declare private structs/classes inside the `CoachingStaff` class if you like. The source files you submit for this project must not contain the word friend. You must not use any global variables whose values may be changed during execution.
- If you wish, you may add a public member function with the signature `void dump() const`. The intent of this function is that for your own testing purposes, you can call it to print information about the map; we will never call it. You do not have to add this function if you don't want to, but if you do add it, it must not make any changes to the map; if we were to replace your implementation of this function with one that simply returned immediately, your code must still work correctly. The dump function must not write to `cout`, but it's allowed to write to `cerr`.
- Your code must build successfully (under both g32 and either clang++ or Visual C++) if linked with a file that contains a main routine.
- You must have an implementation for every member function of `CoachingStaff`, as well as the non-member functions `mergeStaffs` and `searchStaff`. Even if you can't get a function implemented correctly, it must have an implementation that at least builds successfully. For example, if you don't have time to correctly implement `CoachingStaff::fireCoach` or `searchStaff`, say, here are implementations that meet this requirement in that they at least build successfully:

```
bool CoachingStaff::fireCoach(const std::string& fname,
                              const std::string& lname)
{
   return false; // not correct, but at least this code compiles
}

void searchStaff(const std::string& fsearch, const std::string&
   lsearch, const CoachingStaff& csOne, CoachingStaff& csResult)
{
   return; // not correct, but at least this code compiles
}
```

You've probably met this requirement if the following file compiles and links with your code. (This uses magic beyond the scope of CS 32.)

```
#include "CoachingStaff.h"
#include <type_traits>

#define CHECKTYPE(f, t) { auto p = (t)(f); (void)p; }

static_assert(std::is_default_constructible<CoachingStaff>::valu
e,
               "Map must be default-constructible.");

static_assert(std::is_copy_constructible<CoachingStaff>::value,
               "Map must be copy-constructible.");

void ThisFunctionWillNeverBeCalled()
{
   CHECKTYPE(&CoachingStaff::operator=, CoachingStaff&
      (CoachingStaff::*)(const CoachingStaff&));
   CHECKTYPE(&CoachingStaff::noCoaches, bool
      (CoachingStaff::*)() const);
   CHECKTYPE(&CoachingStaff::numberOfCoaches, int
      (CoachingStaff::*)() const);
   CHECKTYPE(&CoachingStaff::hireCoach, bool (CoachingStaff::*)
      (const std::string&, const std::string&, const IType&));
   CHECKTYPE(&CoachingStaff::renameCoach, bool
      (CoachingStaff::*)(const std::string&, const std::string&,
       const IType&));
   CHECKTYPE(&CoachingStaff::hireOrRename, bool
      (CoachingStaff::*)(const std::string&, const std::string&,
       const IType&));
   CHECKTYPE(&CoachingStaff::fireCoach, bool (CoachingStaff::*)
      (const std::string&, const std::string&));
   CHECKTYPE(&CoachingStaff::coachOnStaff, bool
      (CoachingStaff::*)(const std::string&, const std::string&)
       const);
   CHECKTYPE(&CoachingStaff::findCoach, bool (CoachingStaff::*)
      (const std::string&, const std::string&, IType&) const);
   CHECKTYPE(&CoachingStaff::whichCoach, bool (CoachingStaff::*)
      (int, const std::string&, const std::string&, IType&)
       const);
   CHECKTYPE(&CoachingStaff::changeStaff, void
      (CoachingStaff::*)(CoachingStaff&));
```

```
   CHECKTYPE(mergeStaffs,  bool (*)(const CoachingStaff&, const
      CoachingStaff&, CoachingStaff&));
   CHECKTYPE(searchStaff, void (*)(const std::string&,
     const std::string&, const CoachingStaff&, CoachingStaff&));
}

int main()
{}
```

If you add `#include <string>` to `CoachingStaff.h`, have the typedef define `IType` as `std::string`, and link your code to a file containing

```
#include "CoachingStaff.h"
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

void test()
{
   CoachingStaff wgym;

   assert(wgym.hireCoach("Valerie", "Kondos",
      "vkondos@athletics.ucla.edu"));
   assert(wgym.hireCoach("Chris", "Waller",
      "cwaller@athletics.ucla.edu"));
   assert(wgym.numberOfCoaches() == 2);

   string first, last, e;

   assert(wgym.findCoach(0, first, last, e)
      && e == "vkondos@athletics.ucla.edu");
   assert(wgym.findCoach(1, first, last, e)  &&
      (first == "Chris" && e == "cwaller@athletics.ucla.edu"));

   return;
}

int main()
{
   test();
   cout << "Passed all tests" << endl;
   return 0;
}
```

the linking must succeed. When the resulting executable is run, it must write `Passed all tests` to `cout` and nothing else to `cout`.

If we successfully do the above, then make no changes to `CoachingStaff.h` other than to change the typedefs for `CoachingStaff` so that `IType` specifies int, recompile `CoachingStaff.cpp`, and link it to a file containing

```
#include "CoachingStaff.h"
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

void test()
{
   CoachingStaff mwp;

   assert(mwp.hireCoach("Adam", "Wright", 41));
   assert(mwp.hireCoach("Jason", "Falitz", 37));
   assert(mwp.numberOfCoaches() == 2);

   string first, last;
   int a;

   assert(mwp.findCoach(0, first, last, a) && a == 37);
   assert(mwp.findCoach(1, first, last, a) && (first == "Adam"
      && a == 41));

   return;
}

int main()
{
   test();
   cout << "Passed all tests" << endl;
   return 0;
}
```

the linking must succeed. When the resulting executable is run, it must write `Passed all tests` to `cout` and nothing else to `cout`.

During execution, if a client performs actions whose behavior is defined by this spec, your program must not perform any undefined actions, such as dereferencing a null or uninitialized pointer.

Your code in `CoachingStaff.h` and `CoachingStaff.cpp` must not read anything from `cin` and must not write anything whatsoever to `cout`. If you want to print things out for debugging purposes, write to `cerr` instead of `cout`. `cerr` is the standard error destination; items written to it by default go to the screen. When we test your program, we will cause everything written to `cerr` to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish.

Turn It In

By Thursday, July 19th, there will be a link on CCLE that will enable you to turn in your source files and report. You will turn in a zip file containing these files:

- `CoachingStaff.h`. When you turn in this file, the typedefs must specify `string` as the `IType`.
- `CoachingStaff.cpp`. Function implementations should be appropriately commented to guide a reader of the code.
- A file named `report.doc` or `report.docx` (in Microsoft Word format) or `report.txt` (an ordinary text file) that contains:
  - A description of the design of your implementation and why you chose it. (A couple of sentences will probably suffice, perhaps with a picture of a typical List and an empty List. Is your list circular? Does it have a dummy node? What's in your nodes?)
  - A brief description of notable obstacles you overcame.
  - Pseudocode for non-trivial algorithms (e.g., `CoachingStaff::fireCoach` and `mergeStaffs`)
  - A list of test cases that would thoroughly test the functions. Be sure to indicate the purpose of the tests. For example, here's the beginning of a presentation in the form of code:

The tests were performed on a map from strings to doubles

```
  // default constructor
CoachingStaff cfb;
  // For an empty list:
assert(cfb.size() == 0);      // test size
assert(cfb.empty());          // test empty
assert(!cfb.remove("Chip", "Kelly"));  // nothing to erase
```

Even if you do not correctly implement all the functions, you must still list test cases that would test them. Don't lose points by thinking "Well, I didn't implement this function, so I won't bother saying how I would have tested it if I had implemented it."