Homework 4

Time due: 9:00 PM Tuesday, March 1

1. The files Map.h and Map.cpp contain the definition and implementation of Map implemented using a doubly-linked list. A client who wants to use a Map has to change the typedefs in Map.h, and within one source file, cannot have two Maps containing different types.

   Eliminate the typedefs, and change Map to be a class template, so that a client can say

   ```
   #include "Map.h"
   #include <string>
   using std::string;
   ...
   Map<int, double> mid;
   Map<string, int> msi;
   mid.insert(42, -1.25);
   msi.insert("Fred", 123);
   ...
   ```

   Also, change `combine` and `subtract` to be function templates.

   (Hint: Transforming the typedef-based solution is a mechanical task that takes five minutes if you know what needs to be done. What makes this problem non-trivial for you is that you haven't done it before; the syntax for declaring templates is new to you, so you may not get it right the first time.)

   (Hint: Template typename parameters don't have to be named with single letters like `T`; they can be names of your choosing. You might find that by choosing `KeyType`and `ValueType`, you'll have many fewer changes to make.)

(Hint: The Node class nested in the Map class can talk about the template parameters of the Map class; it should not itself be a template class.)

The definitions *and* implementations of your Map class template and the `combine` and `subtract` template functions should be in just one file, Map.h, which is all that you will turn in for this problem. Although the implementation of a non-template non-inline function should not be placed in a header file (because of linker problems if that header file were included in multiple source files), the implementation of a template function, whether or not it's declared inline, *can* be in a header file without causing linker problems.

There's a C++ language technicality that relates to a type declared inside a class template, like `N` below:

```
template <typename T>
class M
{
  ...
  struct N
  {
    ...
  };
  N* f();
  ...
};
```

If we attempt to implement `f` this way:

```
template <typename T>
M<T>::N* M<T>::f()        // Error!  Won't compile.
{
  ...
}
```

the technicality requires the compiler to not recognize `M<T>::N` as a type name; it must be announced as a type name this way:

```
template <typename T>
```

```
typename M<T>::N* M<T>::f()        // OK
{
  ...
}
```

2. Consider this program:

```
#include "Map.h"  // class template from problem 1

   class Coord
   {
     public:
       Coord(int r, int c) : m_r(r), m_c(c) {}
       Coord() : m_r(0), m_c(0) {}
       double r() const { return m_r; }
       double c() const { return m_c; }
     private:
       double m_r;
       double m_c;
   };

int main()
{
    Map<int, double> mid;
    mid.insert(42, -1.25);          // OK
    Map<Coord, int> mpi;
    mpi.insert(Coord(40,10), 32);  // error!
}
```

   Explain in a sentence or two why the call to `Map<Coord, int>::insert` causes at least one compilation error. (Notice that the call to `Map<int, double>::insert` is fine.) Don't just transcribe a compiler error message; your answer must indicate you understand the the ultimate root cause of the problem and why that is connected to the call to `Map<Coord, int>::insert`.

3. A class has a *name* (e.g., `Actor`) and zero or more*subclasses* (e.g., the class with name `Squirt` or the class with name `Protester`). The following program reflects this structure:

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class Class
{
  public:
    Class(string nm) : m_name(nm) {}
    string name() const { return m_name; }
    const vector<Class*>& subclasses() const { return m_subclasses; }
    void add(Class* d) { m_subclasses.push_back(d); }
    ~Class();
  private:
    string m_name;
    vector<Class*> m_subclasses;
};

Class::~Class()
{
    for (size_t k = 0; k < m_subclasses.size(); k++)
        delete m_subclasses[k];
}

void listAll(const Class* c, string path)  // two-parameter overload
{
    You will write this code.
}

void listAll(const Class* c)  // one-parameter overload
{
    if (c != nullptr)
        listAll(c, "");
}

int main()
{
```

```
    Class* d1 = new Class("HardcoreProtester");
    listAll(d1);
    cout << "====" << endl;
    d1->add(new Class("ExtremeProtester"));
    Class* d2 = new Class("Protester");
    d2->add(new Class("RegularProtester"));
    d2->add(d1);
    Class* d3 = new Class("ActivatingObject");
    d3->add(new Class("OilBarrel"));
    d3->add(new Class("GoldNugget"));
    d3->add(new Class("SonarKit"));
    listAll(d3);
    cout << "====" << endl;
    Class* d4 = new Class("Actor");
    d4->add(d2);
    d4->add(new Class("Squirt"));
    d4->add(d3);
    listAll(d4);
    delete d4;
}
```

This main routine should produce the following output (the first line written is `HardcoreProtester`, not an empty line):

```
HardcoreProtester
====
ActivatingObject
ActivatingObject=>OilBarrel
ActivatingObject=>GoldNugget
ActivatingObject=>SonarKit
====
Actor
Actor=>Protester
Actor=>Protester=>RegularProtester
Actor=>Protester=>HardcoreProtester
Actor=>Protester=>HardcoreProtester=>ExtremeProtester
Actor=>Squirt
Actor=>ActivatingObject
Actor=>ActivatingObject=>OilBarrel
```

```
Actor=>ActivatingObject=>GoldNugget
Actor=>ActivatingObject=>SonarKit
```

Each call to the one-parameter overload `listAll` produces a list, one per line, of the inheritance path to each class in the inheritance tree rooted at `listAll`'s argument. An inheritance path is a sequence of class names separated by "=>". There is no "=>" before the first name in the inheritance path.

a. You are to write the code for the two-parameter overload of `listAll` to make this happen. You must not use any additional container (such as a stack), and the two-parameter overload of `listAll` must be recursive. You must not use any global variables or variables declared with the keyword `static`, and you must not modify any of the code we have already written or add new functions. You may use a loop to traverse the vector; you must not use loops to avoid recursion.

   Here's a useful function to know: The standard library string class has a + operator that concatenates strings and/or characters. For example,

```
string s("Hello");
string t("there");
string u = s + ", " + t + '!';
// Now u has the value "Hello, there!"
```

   It's also useful to know that if you choose to traverse an STL container using some kind of iterator, then if the container is const, you must use a `const_iterator`:

```
void f(const list<int>& c)  // c is const
{
    for (list<int>::const_iterator it = c.begin(); it != c.end(); it++)
        cout << *it << endl;
}
```

   (Of course, a vector can be traversed either by using some kind of iterator, or by using operator[] with an integer argument).

For this problem, you will turn a file named `list.cpp` with the body of the two-parameter overload of the `listAll` function, from its "void" to its "}", no more and no less. Your function must compile and work correctly when substituted into the program above.

b. We introduced the two-parameter overload of `listAll`. Why could you not solve this problem given the constraints in part a if we had only a one-parameter `listAll`, and you had to implement *it* as the recursive function?

4.

a. A secretive government agency has reluctantly admitted that for N phone numbers, numbered 0 through N-1, it has a two-dimensional array of bool `hasCommunicatedWith` that records which phones have been in communication with others: `hasCommunicatedWith[i][j]` is true if and only if phone i and phone j have been in communication. If phone i has communicated with phone k, and phone k has communicated with phone j, we call phone k a *direct intermediary* between phone i and phone j.

   The agency has an algorithm that, for every pair of phones i and j, determines how many direct intermediaries they have between them. Here's the code:

```
const int N = some value;
bool hasCommunicatedWith[N][N];
...
int numIntermediaries[N][N];
for (int i = 0; i < N; i++)
{
    numIntermediaries[i][i] = -1;  // the concept of
intermediary
                                   // makes no sense in
this case
    for (int j = 0; j < N; j++)
    {
        if (i == j)
            continue;
```

```
            numIntermediaries[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                if (k == i  ||  k == j)
                    continue;
                if (hasCommunicatedWith[i][k]  &&
hasCommunicatedWith[k][j])
                    numIntermediaries[i][j]++;
            }
        }
    }
```

What is the time complexity of this algorithm, in terms of the number of basic operations (e.g., additions, assignments, comparisons) performed: Is it O(N), O(N log N), or what? Why? (Note: In this homework, whenever we ask for the time complexity, we care only about the high order term, so don't give us answers like $O(N^2+4N)$.)

b. The algorithm in part a doesn't take advantage of the symmetry of communication: for every pair of phones i and j, hasCommunicatedWith[i][j] == hasCommunicatedWith[j][i]. One can skip a lot of operations and compute the number of direct intermediaries more quickly with this algorithm:

```
    const int N = some value;
    bool hasCommunicatedWith[N][N];
    ...
    int numIntermediaries[N][N];
    for (int i = 0; i < N; i++)
    {
        numIntermediaries[i][i] = -1;  // the concept of
intermediary
                                       // makes no sense in
this case
        for (int j = 0; j < i; j++)  // loop limit is now i,
not N
        {
            numIntermediaries[i][j] = 0;
```

```
                for (int k = 0; k < N; k++)
                {
                    if (k == i  ||  k == j)
                        continue;
                    if (hasCommunicatedWith[i][k]  &&
  hasCommunicatedWith[k][j])
                        numIntermediaries[i][j]++;
                }
                numIntermediaries[j][i] =
  numIntermediaries[i][j];
            }
        }
```

What is the time complexity of this algorithm? Why?

5. Here again is the non-member `combine` function for Maps from Map.cpp:

```
bool combine(const Map& m1, const Map& m2, Map& result)
{
    // For better performance, the bigger map should be the basis
for
    // the result, and we should iterate over the elements of the
    // smaller one, adjusting the result as required.

  const Map* bigger;
  const Map* smaller;
  if (m1.size() >= m2.size())
  {
      bigger = &m1;
      smaller = &m2;
  }
  else
  {
      bigger = &m2;
      smaller = &m1;
  }

    // Guard against the case that result is an alias for m1 or m2
```

```
      // (i.e., that result is a reference to the same map that m1 or
m2
      // refers to) by building the answer in a local variable res.
When
      // done, swap res with result; the old value of result (now in
res) will
      // be destroyed when res is destroyed.

    bool status = true;
    Map res(*bigger);              // res starts as a copy of the
bigger map
    for (int n = 0; n < smaller->size(); n++)  // for each pair in
smaller
    {
        KeyType k;
        ValueType vsmall;
        smaller->get(n, k, vsmall);
        ValueType vbig;
        if (!res.get(k, vbig))      // key in smaller doesn't appear
in bigger
            res.insert(k, vsmall);  //     so add it to res
        else if (vbig != vsmall)    // same key, different value
        {                           //     so pair shouldn't be in
res
            res.erase(k);
            status = false;
        }
    }
    result.swap(res);
    return status;
}
```

Assume that `m1`, `m2`, and the old value of `result` each have N elements. In terms of the number of linked list nodes visited during the execution of this function, what is its time complexity? Why?

6. The file sorts.cpp contains an almost complete program that creates a randomly ordered array, sorts it in a few ways, and reports on the elapsed times. Your job is to complete it and experiment with it.

You can run the program as is to get some results for the STL sort algorithm. The insertion sort will give you an assertion violation, because the insertion sort function right now doesn't do anything. That's one thing for you to write.

The objects in the array are not cheap to copy, which makes a sort that does a lot of moving objects around expensive. Your other task will be to create a vector of*pointers* to the objects, sort the pointers using the same criterion as was used to sort the objects, and then make one pass through the vector to put the objects in the proper order.

Your two tasks are thus:

a. Implement the `insertion_sort` function.
b. Implement the `compareStorePtr` function and the code in `main` to create and sort the array of pointers.

The places to make modifications are indicated by `TODO:`comments. You should not have to make modifications anywhere else. (Our solution doesn't.)

Try the program with about 10000 items. Depending on the speed of your processor, this number may be too large or small to get meaningful timings in a reasonable amount of time. Experiment. Once you get insertion sort working, observe the $O(N^2)$ behavior by sorting, say, 10000, 20000, and 40000 items.

To further observe the performance behavior of the STL sort algorithm, try sorting, say, 100000, 200000, and 400000 items, or even ten times as many. Since this would make the insertion sort tests take a long time, skip them by setting the TEST_INSERTION_SORT constant at the top of sorts.cpp to false.

Notice that if you run the program more than once, you may not get the same timings each time. This is *not*because you're not getting the same sequence of random numbers each time; you are. Instead, factors like caching by the operating system are the cause.

Turn it in

By Monday, February 29, 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 four files:

- `Map.h`, a C++ header file with your definition and implementation of the Map class template for problem 1. (Note: You will not turn in a Map.cpp file.) This test program that we will try with your header must build and execute successfully, with no Map.cpp file:

```
#include "Map.h"
#include <cassert>
#include <string>
#include <iostream>

using namespace std;

void test()
{
    Map<int, double> mid;
    Map<string, int> msi;
    assert(msi.empty());
    assert(msi.size() == 0);
    assert(msi.insert("Hello", 10));
    assert(mid.insert(10, 3.5));
    assert(msi.update("Hello", 20));
    assert(mid.update(10, 4.75));
    assert(msi.insertOrUpdate("Goodbye", 30));
    assert(msi.erase("Goodbye"));
    assert(mid.contains(10));
    int k;
    assert(msi.get("Hello", k));
    string s;
    assert(msi.get(0, s, k));
    Map<string, int> msi2(msi);
    msi2.swap(msi);
    msi2 = msi;
    combine(msi,msi2,msi);
    combine(mid,mid,mid);
    subtract(msi,msi2,msi);
```

```
    subtract(mid,mid,mid);
}

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

- `list.cpp`, a C++ source file with the implementation of the two-parameter overload of the `listAll` function for problem 3a.
- `sorts.cpp`, a C++ source file with your solution to problem 6.
- `hw.doc`, `hw.docx`, or `hw.txt`, a Word document or a text file with your solutions to problems 2, 3b, 4a, 4b, and 5.