Winter 2020 CS 32

# Homework 4

**Time due: 11:00 PM Tuesday, March 3**

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 type alias declarations in Map.h, and within one source file, cannot have two Maps containing different types.

   Eliminate the `using` statements defining the type aliases, 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 `reassign` to be function templates.

   (Hint: Transforming the solution based on type aliases 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 the names `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 `reassign` template functions must 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();
     ...
   };
   ```

The technicality affects how we specify the return type of a function (such as `M<T>::f`) when that return type uses a type defined inside a template class (such as `M<T>::N`). 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
{
  ...
}
```

For you to not get a score of zero for this problem, this test program that we will try with your `Map.h` **must**build and execute successfully under both g32 and either Visual C++ or clang++, with no `Map.cpp` file on the command line (for g32) or as part of the project (for Visual C++ or Xcode):

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

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);
    reassign(msi,msi2);
    reassign(mid,mid);
}

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

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. In tracking the spread of a novel virus strain, the Centers for Disease Control and Prevention maintains, for N people numbered 0 through N-1, a two-dimensional array of bool `hasContacted`that records which people have been in contact with others in a way that could transmit the virus: `hasContacted[i][j]` is true if and only if person i and person j have been in such contact. If person i has been in contact with person k, and person k has been in contact with person j, we call person k a *direct intermediary* between person i and person j.

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

```
const int N = some value;
bool hasContacted[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 (hasContacted[i][k]  &&  hasContacted[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 contact: for every pair of persons i and j, hasContacted[i][j] == hasContacted[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 hasContacted[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 (hasContacted[i][k]  &&  hasContacted[k][j])
                numIntermediaries[i][j]++;
        }
        numIntermediaries[j][i] = numIntermediaries[i][j];
    }
}
```

What is the time complexity of this algorithm? Why?

4.

a. Here again is the non-member `reassign` function for Map from Map.cpp:

```
void reassign(const Map& m, Map& result)
{
      // Guard against the case that result is an alias for m (i.e., that
      // result is a reference to the same map that m 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.

    Map res;

    if (!m.empty())
    {
        KeyType prevKey;
        ValueType value0;

          // Get pair 0, which must succeed since m is not empty

        m.get(0, prevKey, value0);

          // For each pair i after pair 0, insert into res a pair with
          // pair i-1's key and pair i's value.  (This loop executes 0 times
          // if m has only one pair.)

        for (int i = 1; i < m.size(); i++)
        {
            KeyType k;
            ValueType v;
            m.get(i, k, v);
            res.insert(prevKey, v);
            prevKey = k;
        }

          // Insert a final pair with last pair's key and pair 0's value.

        res.insert(prevKey, value0);
    }

    result.swap(res);
}
```

Assume that `m` 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?

b. Here is an implementation of a related member function. The call

```
m.reassign();
```

has the same effect as calling the non-member function above as `reassign(m, m);`. The implementation is

```
void Map::reassign()
{
    Node* p = m_head->m_next;
    if (p != m_head)
    {
        ValueType value0 = p->m_value;
        for ( ; p->m_next != m_head; p = p->m_next)
            p->m_value = p->m_next->m_value;
        p->m_value = value0;
    }
}
```

Assume that `*this` has N elements. In terms of the number of linked list nodes visited during the execution of this function, what is its time complexity? Why? Is it the same, better, or worse, than the implementation in part a?

5. 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. You won't get any result for insertion sort, because the insertion sort function right now doesn't do anything. That's one thing for you to write.

   The objects in the array might not be cheap to copy (it depends on your processor), which might make 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 `compareStudentPtr` 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.)

When your program is correct, build an optimized version of it to do some timing experiments. On cs32.seas.ucla.edu, build the executable and run it this way:

```
        g32fast -o sorts sorts.cpp
        ./sorts
```

(You don't have to know this, but this command omits some of the runtime error checking compiler options that our g32 command supplies, and it adds the -O2 compiler option that causes the compiler to spend more time optimizing the machine language translation of your code so that it will run faster when you execute it.)

Under Xcode, select Product / Scheme / Edit Scheme.... In the left panel, select Run, then in the right panel, select the Info tab. In the Build Configuration dropdown, select Release. For Visual C++, it's a little trickier.

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 exactly the same timings each time. This is partly because of not getting the same sequence of random numbers each time, but also because of factors like caching by the operating system.

## Turn it in

By Monday, March 2, 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 three files:

- `Map.h`, a C++ header file with your definition and implementation of the class and function templates for problem 1.
- `sorts.cpp`, a C++ source file with your solution to problem 5.
- `hw.docx`, `hw.doc`, or `hw.txt`, a Word document or a text file with your solutions to problems 2, 3a, 3b, 4a, and 4b.