Homework 4

Time due: 9:00 PM Tuesday, March 3

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

   Change Multiset to be a class template, so that a client can say

   ```
   #include "Multiset.h"
   #include <string>
   using std::string;
   ...
   Multiset<int> msi;
   Multiset<string> mss;
   msi.insert(5);
   mss.insert("Maroon 5");
   ...
   ```

   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 `Itemtype` you'll have many fewer changes to make.)

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

   The definition *and* implementation of your Multiset class template should be in just one file, Multiset.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.

   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!
{
  ...
}
```

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 "Multiset.h"  // class template from problem 1
#include <string>
using namespace std;

class URL
{
  public:
    URL(string i) : m_id(i) {}
    URL() : m_id("http://cs.ucla.edu") {}
    string id() const { return m_id; }
  private:
    string m_id;
};

int main()
{
    Multiset<int> mi;
    mi.insert(7);  // OK
    Multiset<string> ms;
    ms.insert("http://www.symantec.com");  // OK
    Multiset<URL> mu;
    mu.insert(URL("http://www.symantec.com"));  // error!
}
```

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

3. Many applications have menus organized in a hierarchical fashion. For example, the menu bar may have File, Edit, and Help menu items. These items may have submenus, some of which may have submenus, etc. Every menu item has a name. When describing the full path to a menu item, we separate levels with slashes (e.g., "File/New/Window"). The following program reflects this structure:

```
#include <iostream>
```

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

using namespace std;

class MenuItem
{
  public:
    MenuItem(string nm) : m_name(nm) {}
    virtual ~MenuItem() {}
    string name() const { return m_name; }
    virtual bool add(MenuItem* m) = 0;
    virtual const vector<MenuItem*>* menuItems() const = 0;
  private:
    string m_name;
};

class PlainMenuItem : public MenuItem   // PlainMenuItem allows no submenus
{
  public:
    PlainMenuItem(string nm) : MenuItem(nm) {}
    virtual bool add(MenuItem* m) { return false; }
    virtual const vector<MenuItem*>* menuItems() const { return NULL; }
};

class CompoundMenuItem : public MenuItem  // CompoundMenuItem allows submenus
{
  public:
    CompoundMenuItem(string nm) : MenuItem(nm) {}
    virtual ~CompoundMenuItem();
    virtual bool add(MenuItem* m) { m_menuItems.push_back(m); return true; }
    virtual const vector<MenuItem*>* menuItems() const { return &m_menuItems; }
  private:
    vector<MenuItem*> m_menuItems;
};

CompoundMenuItem::~CompoundMenuItem()
{
    for (int k = 0; k < m_menuItems.size(); k++)
        delete m_menuItems[k];
}

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

void listAll(const MenuItem* m)  // one-parameter overload
{
    if (m != NULL)
        listAll(m, "");
}

int main()
{
    CompoundMenuItem* cm0 = new CompoundMenuItem("New");
    cm0->add(new PlainMenuItem("Window"));
    CompoundMenuItem* cm1 = new CompoundMenuItem("File");
    cm1->add(cm0);
```

```
    cm1->add(new PlainMenuItem("Open"));
    cm1->add(new PlainMenuItem("Exit"));
    CompoundMenuItem* cm2 = new CompoundMenuItem("Help");
    cm2->add(new PlainMenuItem("Index"));
    cm2->add(new PlainMenuItem("About"));
    CompoundMenuItem* cm3 = new CompoundMenuItem("");  // main menu bar
    cm3->add(cm1);
    cm3->add(new PlainMenuItem("Refresh"));  // no submenu
    cm3->add(new CompoundMenuItem("Under Development")); // no submenus yet
    cm3->add(cm2);
    listAll(cm3);
    delete cm3;
}
```

When the listAll function is called from the main routine above, the following output should be produced (the first line written is `File`, not an empty line):

```
File
File/New
File/New/Window
File/Open
File/Exit
Refresh
Under Development
Help
Help/Index
Help/About
```

Each call to the one-parameter overload `listAll` produces a list, one per line, of the complete path to each menu item in the tree rooted at `listAll`'s argument. A path is a sequence of menu item names separated by "/". There is no "/" before the first name in the 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. Suppose we have a list of N world cities, numbered from 0 to N-1. The two-dimensional array of doubles `dist` holds the airline distance between each pair of cities: `dist[i][j]` is the distance between city i and city j.

Now, for every pair of cities i and j, we'd like to consider all the flights between the two that make one stop in a third city k, and record which city k yields the shortest distance traveled in a one-stop flight between city i and city j that passes through city k. Here's the code:

```
const int N = some value;
assert(N > 2);  // algorithm fails if N <= 2
double dist[N][N];
...
int bestMidPoint[N][N];
for (int i = 0; i < N; i++)
{
    bestMidPoint[i][i] = -1;  // one-stop trip to self is silly
    for (int j = 0; j < N; j++)
    {
        if (i == j)
            continue;
        int minDist = maximum possible integer;
        for (int k = 0; k < N; k++)
        {
            if (k == i  ||  k == j)
                continue;
            int d = dist[i][k] + dist[k][j];
            if (d < minDist)
            {
                minDist = d;
                bestMidPoint[i][j] = k;
            }
        }
    }
}
```

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 distances: for every pair of cities i and j, dist[i][j] == dist[j][i]. We can skip a lot of operations and compute the best midpoints more quickly with this algorithm:

```
const int N = some value;
assert(N > 2);  // algorithm fails if N <= 2
double dist[N][N];
...
int bestMidPoint[N][N];
for (int i = 0; i < N; i++)
{
    bestMidPoint[i][i] = -1;  // one-stop trip to self is silly
    for (int j = 0; j < i; j++)  // loop limit is now i, not N
    {
        int minDist = maximum possible integer;
        for (int k = 0; k < N; k++)
        {
            if (k == i  ||  k == j)
                continue;
            int d = dist[i][k] + dist[k][j];
            if (d < minDist)
            {
                minDist = d;
                bestMidPoint[i][j] = k;
                bestMidPoint[j][i] = k;
            }
        }
    }
}
```

What is the time complexity of this algorithm? Why?

5. a. Consider the Multiset class defined in the solution to Project 2. Here is a function to produce in `result` exactly one instance of each item that appears in both `m1` and `m2`, regardless of the number of repeated occurrences in those multisets:

```
void uniqueIntersect(const Multiset& m1, const Multiset& m2, Multiset& result)
{
    Multiset res;
    for (int k = 0; k != m1.uniqueSize(); k++)
    {
        ItemType x;
        m1.get(k, x);
        if (m2.contains(x))
            res.insert(x);
    }
    result.swap(res);
}
```

Assume that m1, m2, and the old value of result each have N distinct items. In terms of the number of ItemType objects visited (in the linked list nodes) during the execution of this function, what is its time complexity? Why?

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

```
m3.uniqueIntersect(m1, m2);
```

produces in `m3` exactly one instance of each item that appears in both `m1` and `m2`. The implementation is

```
void Multiset::uniqueIntersect(const Multiset& m1, const Multiset& m2)
```

```
{
    vector<ItemType> v;
    v.reserve(m1.uniqueSize() + m2.uniqueSize());

      // copy all items into v;
    for (Node* p1 = m1.m_head->m_next; p1 != m1.m_head; p1 = p1->m_next)
        v.push_back(p1->m_data);
    for (Node* p2 = m2.m_head->m_next; p2 != m2.m_head; p2 = p2->m_next)
        v.push_back(p2->m_data);

      // sort v using an O(N log N) algorithm
    sort(v.begin(), v.end());

      // Items in the intersection will be those that appear twice in
      // v, adjacent to each other.
      // Copy one instance of those items from v into *this.
    m_size = 0;
    Node* p = m_head->next;
    for (size_t k = 1; k < v.size(); k++)
    {
        if (v[k] == v[k-1])
        {
            Node* toUpdate;
            if (p != m_head)
            {
                toUpdate = p;  // reuse existing node
                p = p->m_next;
            }
            else
            {
                  // Insert new node at tail of result
                toUpdate = new Node;
                toUpdate->m_next = m_head;
                toUpdate->m_prev = m_head->m_prev;
                toUpdate->m_prev->m_next = toUpdate;
                toUpdate->m_next->m_prev = toUpdate;
            }
            toUpdate->m_value = v[k];
            toUpdate->m_count = 1;
            m_size++;
        }
    }

      // delete excess result nodes
    if (p != m_head)
    {
        m_head->m_prev = p->m_prev;
        p->m_prev->m_next = m_head;
        do
        {
            Node* toBeDeleted = p;
            p = p->m_next;
            delete toBeDeleted;
        } while (p != m_head);
    }

    m_uniqueSize = m_size;

      // v is destroyed when function returns
```

```
}
```

Assume that `m1`, `m2`, and the old value of `*this` each have N distinct items. In terms of the number of ItemType objects visited (either in linked list nodes or the vector) during the execution of this function, what is its average case time complexity? Why? Is it the same, better, or worse, than the implementation in part a?

(Note: `vector<T>::reserve` is constant time when called on an empty vector, and `vector<T>::operator[]` is constant time. Calling `vector<T>::reserve` the way we did ensures that the vector has the capacity to add each value without having to reallocate its dynamic array.)

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 result will be meaningless (and you'll get an assertion violation), because the insertion sort function right now doesn't do anything. That's one thing for you to write. (Note: The insertion sort algorithm on pp. 312-313 of the Carrano book 6th edition is incorrect; someone made a change from the 5th edition and messed it up. See the errata page entry for page 313.)

   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 `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.)

   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 30000 items.

   To further observe the performance behavior of the STL sort algorithm, try sorting, say, 100000, 200000, and 300000 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 *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, 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 four files:

- `Multiset.h`, a C++ header file with your definition and implementation of the Multiset class template for problem 1. This test program that we will try with your header must build and execute successfully:

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

void test()
{
    Multiset<int> mi;
    Multiset<string> ms;
    assert(mi.empty());
    assert(ms.size() == 0);
    assert(mi.uniqueSize() == 0);
    assert(mi.insert(10));
    assert(ms.insert("hello"));
    assert(mi.contains(10));
    assert(ms.count("hello") == 1);
    assert(mi.erase(10) == 1);
    string s;
    assert(ms.get(0, s)  &&  s == "hello");
    Multiset<string> ms2(ms);
    ms.swap(ms2);
    ms = ms2;
    combine(mi,mi,mi);
    combine(ms,ms2,ms);
    assert(ms.eraseAll("hello") == 2);
    subtract(mi,mi,mi);
    subtract(ms,ms2,ms);
}

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, 5a, and 5b.