Homework 1

Time due: 9:00 PM Tuesday, January 20

Here is a C++ class definition for an abstract data type **Multiset of `string`s**, representing the concept of an unordered collection of strings with duplicates allowed. (A web server might record visits to a website in a multiset of strings, for example: Each time someone visits the site, a string identifying the visitor is stored in the multiset. The same visitor may visit multiple times.) To make things simpler for you, the case of letters in a string matters, so that the strings `cumin` and `cUmIn` are *not* considered duplicates.

```
class Multiset
{
  public:
    Multiset();    // Create an empty multiset.

    bool empty();  // Return true if the multiset is empty, otherwise false.

    int size();
      // Return the number of items in the multiset.  For example, the size
      // of a multiset containing "cumin", "cumin", "cumin", "turmeric" is 4.

    int uniqueSize();
      // Return the number of distinct items in the multiset.  For example,
      // the uniqueSize of a multiset containing "cumin", "cumin", "cumin",
      // "turmeric" is 2.

    bool insert(const std::string& value);
      // Insert value into the multiset.  Return true if the value was
      // actually inserted.  Return false if the value was not inserted
      // (perhaps because the multiset has a fixed capacity and is full).

    int erase(const std::string& value);
      // Remove one instance of value from the multiset if present.
      // Return the number of instances removed, which will be 1 or 0.

    int eraseAll(const std::string& value);
      // Remove all instances of value from the multiset if present.
      // Return the number of instances removed.

    bool contains(const std::string& value);
      // Return true if the value is in the multiset, otherwise false.

    int count(const std::string& value);
      // Return the number of instances of value in the multiset.

    int get(int i, std::string& value);
      // If 0 <= i < uniqueSize(), copy into value an item in the
      // multiset and return the number of instances of that item in
      // the multiset.  Otherwise, leave value unchanged and return 0.
      // (See below for details about this function.)

    void swap(Multiset& other);
```

```
        // Exchange the contents of this multiset with the other one.
};
```

(When we don't want a function to change a parameter representing a value of the type stored in the multiset, we pass that parameter by constant reference. Passing it by value would have been perfectly fine for this problem, but we chose the const reference alternative because that will be more suitable after we make some generalizations in a later problem.)

The `get` function enables a client to iterate over all elements of a `Multiset` because of this property it must have: If nothing is inserted into or erased from the multiset in the interim, then calling `get` repeatedly with the first parameter ranging over all the integers from `0` to `uniqueSize()-1` inclusive will copy into the second parameter every distinct value from the multiset exactly once. The order in which elements are copied is up to you. In other words, this code fragment

```
Multiset ms;
ms.insert("fennel");
ms.insert("fennel");
ms.insert("fenugreek");
ms.insert("fennel");
for (int k = 0; k < ms.uniqueSize(); k++)
{
    string x;
    int n = ms.get(k, x);
    cout << x << " occurs " << n << " times." << endl;
}
```

must write either

```
fennel occurs 3 times.
fenugreek occurs 1 times.
```

or

```
fenugreek occurs 1 times.
fennel occurs 3 times.
```

and the client can't depend on it being any particular one of those two. If the multiset is modified between successive calls to `get`, all bets are off as to whether a particular value is returned exactly once.

If nothing is inserted into or erased from the multiset in the interim, then calling `get` repeatedly with the same value for the first parameter each time must copy the same value into the second parameter each time and return the same value each time, so that this code is fine:

```
Multiset ms;
ms.insert("cinnamon");
ms.insert("galangal");
ms.insert("cinnamon");
string s1;
int n1 = ms.get(1, s1);
assert((s1 == "cinnamon"  &&  n1 == 2)  ||  (s1 == "galangal"  &&  n1 == 1));
string s2;
int n2 = ms.get(1, s2);
assert(s2 == s1  &&  n2 == n1);
```

Here's an example of the `swap` function:

```
Multiset ms1;
ms1.insert("cumin");
ms1.insert("cumin");
ms1.insert("cumin");
ms1.insert("turmeric");
Multiset ms2;
ms2.insert("coriander");
ms2.insert("cumin");
ms2.insert("cardamom");
ms1.swap(ms2);  // exchange contents of ms1 and ms2
assert(ms1.size() == 3  &&  ms1.count("coriander") == 1  &&
           ms1.count("cumin") == 1  &&  ms1.count("cardamom") == 1);
assert(ms2.size() == 4  &&  ms2.count("cumin") == 3  &&
           ms2.count("turmeric") == 1);
```

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

```
Multiset ms;
ms.insert("cumin");
assert(!ms.contains(""));
ms.insert("nutmeg");
ms.insert("");
ms.insert("saffron");
assert(ms.contains(""));
ms.erase("cumin");
assert(ms.size() == 3  &&  ms.contains("saffron")  &&  ms.contains("nutmeg")  &&
           ms.contains(""));
```

When comparing items to see if they're duplicates, just use the `==` or `!=` operators provided for the string type by the library. These do case-sensitive comparisons, and that's fine.

Here is what you are to do:

1. Determine which member functions of the Multiset class should be const member functions (because they do not modify the Multiset), and change the class declaration accordingly.

2. As defined above, the Multiset class allows the client to use a multiset that contains only `std::string`s. Someone who wanted to modify the class to contain items of another type, such as only `int`s or only `double`s, would have to make changes in many places. Modify the class definition you produced in the previous problem to use a `typedef`-defined type for all values wherever the original definition used a `std::string`. Here's an example of a use of `typedef`:

```
typedef int Number;  // define Number as a synonym for int

int main()
{
    Number total = 0;
    Number x;
    while (cin >> x)
        total += x;
    cout << total << endl;
}
```

To modify this code to sum a sequence of `long`s or of `double`s, we need make a change in only one place: the typedef.

You may find the example using `typedef` in Appendix A.1.8 of the textbook useful.

To make the grader's life easier, we'll require that everyone use the same synonym as their `typedef`-defined name: You must use the name `ItemType`, with exactly that spelling and case.

3. Now that you have defined an interface for a multiset class where the item type can be easily changed, implement the class and all its member functions in such a way that the items in a multiset are contained in a data member that is an array. (Notice we said an array, not a pointer. It's not until problem 5 of this homework that you'll deal with a dynamically allocated array.) A multiset must be able to hold a maximum of `DEFAULT_MAX_ITEMS` distinct items, where

```
const int DEFAULT_MAX_ITEMS = 200;
```

(Hint: Define a structure type containing a member of type `ItemType` and a member of type `int` (representing a count). Have Multiset's array data member be an array of these structures.)

Test your class for a Multiset of `unsigned long`s. Place your class definition and inline function definitions (if any) in a file named `Multiset.h`, and your non-inline function definitions (if any) in a file named `Multiset.cpp`.

You may add any private data members or private member functions that you like, but you must not add anything to or delete anything from the public interface you defined in the previous problem, nor may you change the function signatures. There is one exception to this: 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 multiset; 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 multiset; 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 implementation of the Multiset class must be such that the compiler-generated destructor, copy constructor, and assignment operator do the right things. Write a test program named `testMultiset.cpp` to make sure your Multiset class implementation works properly. Here is one possible (incomplete) test program:

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

int main()
{
    Multiset ms;
    assert(ms.empty());
    unsigned long x = 999;
    assert(ms.get(0, x) == 0  &&  x == 999);  // x unchanged by get failure
    assert( ! ms.contains(42));
    ms.insert(42);
    ms.insert(42);
```

```
    assert(ms.size() == 2  &&  ms.uniqueSize() == 1);
    assert(ms.get(1, x) == 0  &&  x == 999);  // x unchanged by get failure
    assert(ms.get(0, x) == 2  &&  x == 42);
    cout << "Passed all tests" << endl;
}
```

Now change (only) the typedef in `Multiset.h` so that the Multiset will now contain `std::string`s. Make no other changes to `Multiset.h`, and make no changes to `Multiset.cpp`. Verify that your implementation builds correctly and works properly with this alternative main routine (which again, is not a complete test of correctness):

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

int main()
{
    Multiset ms;
    assert(ms.empty());
    string x = "dill";
    assert(ms.get(0, x) == 0  &&  x == "dill");  // x unchanged by get failure
    assert( ! ms.contains("tamarind"));
    ms.insert("tamarind");
    ms.insert("tamarind");
    assert(ms.size() == 2  &&  ms.uniqueSize() == 1);
    assert(ms.get(1, x) == 0  &&  x == "dill");  // x unchanged by get failure
    assert(ms.get(0, x) == 2  &&  x == "tamarind");
    cout << "Passed all tests" << endl;
}
```

You may need to flip back and forth a few times to fix your `Multiset.h` and `Multiset.cpp` code so that the *only* change to those files you'd need to make to change a multiset's item type is to the typedef in `Multiset.h`. (When you turn in the project, have them so that the item type is `unsigned long`.)

(Implementation note 1: If you declare another structure to help you implement a Multiset, put its declaration in `Multiset.h` (and `newMultiset.h` for Problem 5), since it is not intended to be used by clients for its own sake, but merely to help you implement the Multiset class. In fact, to enforce clients' not using that structure type, don't declare it outside of the Multiset class; instead declare that helper structure in the private section of Multiset. Although it would probably be overkill for this structure to have anything more than two public data members, if for some reason you decide to declare any member functions for it that need to be implemented, those implementations should be in `Multiset.cpp` (and `newMultiset.cpp` for Problem 5).)

Except in the typedef statement in `Multiset.h`, the words `unsigned` and `long` must not appear in `Multiset.h` or `Multiset.cpp`. Except in the context of `#include <string>`, the word `string` must not appear in `Multiset.h` or `Multiset.cpp`.

(Implementation note 2: The `swap` function is easily implementable without creating any additional array or additional Multiset.)

4. Now that you've implemented the class, write some client code that uses it. We might want a class that

records all CS 32 student project submissions. Students may make more than one submission. Implement the following class that uses a Multiset of `unsigned long`s:

```
#include "Multiset.h"

class StudentMultiset
{
  public:
    StudentMultiset();       // Create an empty student multiset.

    bool add(unsigned long id);
      // Add a student id to the StudentMultiset.  Return true if and only
      // if the id was actually added.

    int size() const;
      // Return the number of items in the StudentMultiset.  If an id was
      // added n times, it contributes n to the size.

    void print() const;
      // Print to cout every student id in the StudentMultiset one per line;
      // print as many lines for each id as it occurs in the StudentMultiset.

  private:
      // Some of your code goes here.
};
```

Your StudentMultiset implementation must employ a data member of type Multiset that uses the typedef `ItemType` as a synonym for `unsigned long`. (Notice we said a member of type *Multiset*, not of type pointer to Multiset.) Except to change one line (the typedef in Multiset.h), you must not make any changes to the `Multiset.h` and `Multiset.cpp` files you produced for Problem 3, so you must not add any member functions to the Multiset class. Each of the member functions `add`, `size`, and `print` must delegate as much of the work that they need to do as they can to Multiset member functions. (In other words, they must not do work themselves that they can ask Multiset member functions to do.) If the compiler-generated destructor, copy constructor, and assignment operator for StudentMultiset don't do the right thing, declare and implement them. Write a program to test your StudentMultiset class. Name your files `StudentMultiset.h`, `StudentMultiset.cpp`, and `testStudentMultiset.cpp`.

The words `for` and `while` must not appear in `StudentMultiset.h` or `StudentMultiset.cpp`, except in the implementation of `StudentMultiset::print` if you wish. The characters `[` (open square bracket) and `*` must not appear in `StudentMultiset.h` or `StudentMultiset.cpp`, except in comments if you wish. You do not have to change `unsigned long` to `ItemType` in `StudentMultiset.h` and `StudentMultiset.cpp` if you don't want to. In the code you turn in, `StudentMultiset`'s member functions must not call `Multiset::dump`.

5. Now that you've created a multiset type based on arrays whose size is fixed at compile time, let's change the implementation to use a *dynamically allocated* array of objects. Copy the three files you produced for problem 3, naming the new files `newMultiset.h`, `newMultiset.cpp`, and `testnewMultiset.cpp`. Update those files by either adding another constructor or modifying your existing constructor so that a client can do the following:

```
Multiset a(1000);   // a can hold at most 1000 distinct items
Multiset b(5);      // b can hold at most 5 distinct items
```

```
Multiset c;         // c can hold at most DEFAULT_MAX_ITEMS distinct items
ItemType v[6] = { six distinct values of the appropriate type };
  // No failures inserting 5 distinct items twice each into b
for (int k = 0; k < 5; k++)
{
    assert(b.insert(v[k]));
    assert(b.insert(v[k]));
}
assert(b.size() == 10  &&  b.uniqueSize() == 5  &&  b.count(v[0]) == 2);
  // Failure if we try to insert a sixth distinct item into b
assert(!b.insert(v[5]));

  // When two Multisets' contents are swapped, their capacities are swapped
  // as well:
a.swap(b);
assert(!a.insert(v[5])  &&  b.insert(v[5]));
```

Since the compiler-generated destructor, copy constructor, and assignment operator no longer do the right thing, declare them (as public members) and implement them. Make no other changes to the public interface of your class. (You are free to make changes to the private members and to the implementations of the member functions.) Change the implementation of the `swap` function so that the number of statement executions when swapping two multisets is the same no matter how many items are in the multisets. (You would not satisfy this requirement if, for example, your swap function looped over each item in a multiset, since the number of iterations of the loop would depend on the number of items in the multiset.)

The character `[` (open square bracket) must not appear in `newMultiset.h` (but is fine in `newMultiset.cpp`).

Test your new implementation of the Multiset class. (Notice that even though the file is named `newMultiset.h`, the name of the class defined therein must still be `Multiset`.)

Verify that your StudentMultiset class still works properly with this new version of Multiset. You should not need to change your StudentMultiset class or its implementation in any way, other than to #include "newMultiset.h" instead of "Multiset.h". (For this test, be sure to link with `newMultiset.cpp`, not `Multiset.cpp`.) (Before you turn in `StudentMultiset.h`, be sure to restore the #include to "Multiset.h" instead of "newMultiset.h".)

Turn it in

By Monday, January 19, 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. (Since problem 3 builds on problems 1 and 2, you will not turn in separate code for problems 1 and 2.) If you solve every problem, the zip file you turn in will have nine files (three for each of problems 3, 4, and 5). The files *must* meet these requirements, or your score on this homework will be severely reduced:

- Each of the header files `Multiset.h`, `StudentMultiset.h`, and `newMultiset.h` must have an appropriate include guard. In the files you turn in, the typedefs in `Multiset.h` and `newMultiset.h` must define `ItemType` to be a synonym for `unsigned long`.

- If we create a project consisting of `Multiset.h`, `Multiset.cpp`, and `testMultiset.cpp`, it must build

successfully under both Visual C++ and either clang++ or g++.

- If we create a project consisting of `Multiset.h`, `Multiset.cpp`, `StudentMultiset.h`, `StudentMultiset.cpp`, and `testStudentMultiset.cpp`, it must build successfully under both Visual C++ and either clang++ or g++.

- If we create a project consisting of `newMultiset.h`, `newMultiset.cpp`, and `testnewMultiset.cpp`, it must build successfully under both Visual C++ and either clang++ or g++.

- If we create a project consisting of `newMultiset.h`, `newMultiset.cpp`, and `testMultiset.cpp`, where in `testMultiset.cpp` we change only the `#include "Multiset.h"` to `#include "newMultiset.h"`, the project must build successfully under both Visual C++ and either clang++ or g++. (If you try this, be sure to change the `#include` back to `"Multiset.h"` before you turn in `testMultiset.h`.)

- The source files you submit for this homework must not contain the word `friend` or `vector`, and must not contain any global variables whose values may be changed during execution. (Global *constants* are fine.)

- No files other than those whose names begin with `test` may contain code that reads anything from `cin` or writes anything to `cout`, except that for problem 4, `StudentMultiset::print` must write to `cout`, and for problem 5, the implementation of the constructor that takes an integer parameter may write a message and exit the program if the integer is negative. Any file may write to `cerr` (perhaps for debugging purposes); we will ignore any output written to `cerr`.

- You must have an implementation for every member function of Multiset and StudentMultiset. If you can't get a function implemented correctly, its implementation must at least build successfully. For example, if you don't have time to correctly implement `Multiset::insert` or `Multiset::swap`, say, here are implementations that meet this requirement in that they at least allow programs to build successfully even though they might execute incorrectly:

```
int Multiset::erase(const ItemType& value)
{
    return -42;  // not correct, but at least this compiles
}

void Multiset::swap(Multiset& other)
{
    // does nothing; not correct, but at least this compiles
}
```

- Given `Multiset.h` with the typedef for the Multiset's item type specifying `std::string`, if we make no change to your `Multiset.cpp`, then if we compile your `Multiset.cpp` and link it to a file containing

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

void test()
{
    Multiset sms;
```

```
    assert(sms.insert("cumin"));
    assert(sms.insert("turmeric"));
    assert(sms.insert("cumin"));
    assert(sms.insert("coriander"));
    assert(sms.insert("cumin"));
    assert(sms.insert("turmeric"));
    assert(sms.size() == 6  &&  sms.uniqueSize() == 3);
    assert(sms.count("turmeric") == 2);
    assert(sms.count("cumin") == 3);
    assert(sms.count("coriander") == 1);
    assert(sms.count("cardamom") == 0);
}

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

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

- If we successfully do the above, then in `Multiset.h` change the Multiset's typedef to specify `unsigned long` as the item type without making any other changes, recompile `Set.cpp`, and link it to a file containing

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

void test()
{
    Multiset ulms;
    assert(ulms.insert(20));
    assert(ulms.insert(10));
    assert(ulms.insert(20));
    assert(ulms.insert(30));
    assert(ulms.insert(20));
    assert(ulms.insert(10));
    assert(ulms.size() == 6  &&  ulms.uniqueSize() == 3);
    assert(ulms.count(10) == 2);
    assert(ulms.count(20) == 3);
    assert(ulms.count(30) == 1);
    assert(ulms.count(40) == 0);
}

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

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

- Given `newMultiset.h` with the typedef for the Multiset's item type specifying `std::string`, if we make

no change to your `newMultiset.cpp`, then if we compile your `newMultiset.cpp` and link it to a file containing

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

void test()
{
    Multiset sms;
    assert(sms.insert("cumin"));
    assert(sms.insert("turmeric"));
    assert(sms.insert("cumin"));
    assert(sms.insert("coriander"));
    assert(sms.insert("cumin"));
    assert(sms.insert("turmeric"));
    assert(sms.size() == 6  &&  sms.uniqueSize() == 3);
    assert(sms.count("turmeric") == 2);
    assert(sms.count("cumin") == 3);
    assert(sms.count("coriander") == 1);
    assert(sms.count("cardamom") == 0);
}

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

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

- If we successfully do the above, then change the typedef for the Multiset's item type to specify `unsigned long` as the item type without making any other changes, recompile `newMultiset.cpp`, and link it to a file containing

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

void test()
{
    Multiset ulms;
    assert(ulms.insert(20));
    assert(ulms.insert(10));
    assert(ulms.insert(20));
    assert(ulms.insert(30));
    assert(ulms.insert(20));
    assert(ulms.insert(10));
    assert(ulms.size() == 6  &&  ulms.uniqueSize() == 3);
    assert(ulms.count(10) == 2);
    assert(ulms.count(20) == 3);
    assert(ulms.count(30) == 1);
```

```
    assert(ulms.count(40) == 0);
}

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

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

- During execution, your program must not perform any undefined actions, such as accessing an array element out of bounds, or dereferencing a null or uninitialized pointer.

Notice that we are not requiring any particular content in `testMultiset.cpp`, `testStudentMultiset.cpp`, and `testnewMultiset.cpp`, as long as they meet the requirements above. Of course, the intention is that you'd use those files for the test code that you'd write to convince yourself that your implementations are correct. Although we will throughly evaluate your implementations for correctness, for homeworks, unlike for projects, we will not grade the thoroughness of your test cases. Incidentally, for homeworks, unlike for projects, we will also not grade your program commenting.