Winter 2020 CS 32

Homework 1

Time due: 11:00 PM Tuesday, January 21

Here is a C++ class definition for an abstract data type **Map from `string` to `double`**, representing the concept of a function mapping `string`s to `double`s. (For example, we could represent a collection of students and their GPAs: "Fred" maps to 2.956, "Ethel" maps to 3.538, "Lucy" maps to 2.956, etc.) We'll call the `string`s the *keys* and the `double`s the *values*. No two keys in the map are allowed to be the same (e.g., so "Fred" appears no more than once), although two keys might map to the same value (as in the example, where both of the keys "Fred" and "Lucy" map to the value 2.956). To make things simpler for you, the case of letters in a string matters, so that the strings `Fred` and `fReD` are *not* considered duplicates.

```
class Map
{
  public:
    Map();         // Create an empty map (i.e., one with no key/value pairs)

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

    int size();    // Return the number of key/value pairs in the map.

    bool insert(const std::string& key, const double& value);
      // If key is not equal to any key currently in the map, and if the
      // key/value pair can be added to the map, then do so and return true.
      // Otherwise, make no change to the map and return false (indicating
      // that either the key is already in the map, or the map has a fixed
      // capacity and is full).

    bool update(const std::string& key, const double& value);
      // If key is equal to a key currently in the map, then make that key no
      // longer map to the value it currently maps to, but instead map to
      // the value of the second parameter; return true in this case.
      // Otherwise, make no change to the map and return false.

    bool insertOrUpdate(const std::string& key, const double& value);
      // If key is equal to a key currently in the map, then make that key no
      // longer map to the value that it currently maps to, but instead map to
      // the value of the second parameter; return true in this case.
      // If key is not equal to any key currently in the map and if the
      // key/value pair can be added to the map, then do so and return true.
      // Otherwise, make no change to the map and return false (indicating
      // that the key is not already in the map and the map has a fixed
      // capacity and is full).

    bool erase(const std::string& key);
      // If key is equal to a key currently in the map, remove the key/value
      // pair with that key from the map and return true.  Otherwise, make
      // no change to the map and return false.

    bool contains(const std::string& key);
      // Return true if key is equal to a key currently in the map, otherwise
      // false.

    bool get(const std::string& key, double& value);
      // If key is equal to a key currently in the map, set value to the
      // value in the map which that key maps to, and return true.  Otherwise,
      // make no change to the value parameter of this function and return
      // false.

    bool get(int i, std::string& key, double& value);
      // If 0 <= i < size(), copy into the key and value parameters the
      // key and value of one of the key/value pairs in the map and return
      // true.  Otherwise, leave the key and value parameters unchanged and
```

```
      // return false.  (See below for details about this function.)

    void swap(Map& other);
      // Exchange the contents of this map with the other one.
};
```

(When we don't want a function to change a key or value parameter, we pass that parameter by constant reference. Passing it by value would have been perfectly fine for this problem, but we're requiring you to use the const reference alternative because that will be more suitable after we make some generalizations in a later problem.)

The three-argument `get` function enables a client to iterate over all elements of a `Map` because of this property it must have: If nothing is inserted into or erased from the map in the interim, then calling that version of `get` size() times with the first parameter ranging over each of the integers from 0 to size()-1 inclusive will copy into the other parameters every key/value pair from the map exactly once. The order in which key/value pairs are copied is up to you. In other words, this client code fragment

```
Map m;
m.insert("A", 10);
m.insert("B", 44);
m.insert("C", 10);
string all;
double total = 0;
for (int n = 0; n < m.size(); n++)
{
    string k;
    double v;
    m.get(n, k, v);
    all += k;
    total += v;
}
cout << all << total;
```

must result in the output being exactly one of the following: `ABC64`, `ACB64`, `BAC64`, `BCA64`, `CAB64`, or `CBA64`, and the client can't depend on it being any particular one of those six. If the map is modified between successive calls to the three-argument form of `get`, all bets are off as to whether a particular key/value pair in the map is returned exactly once.

If nothing is inserted into or erased from the map in the interim, then calling the three-argument form of `get` repeatedly with the same value for the first parameter each time must copy the same key into the second parameter each time and the same value into the third parameter each time, so that this code is fine:

```
Map gpas;
gpas.insert("Fred", 2.956);
gpas.insert("Ethel", 3.538);
double v;
string k1;
assert(gpas.get(1,k1,v)  &&  (k1 == "Fred"  ||  k1 == "Ethel"));
string k2;
assert(gpas.get(1,k2,v)  &&  k2 == k1);
```

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

```
Map gpas;
gpas.insert("Fred", 2.956);
assert(!gpas.contains(""));
gpas.insert("Ethel", 3.538);
gpas.insert("", 4.000);
gpas.insert("Lucy", 2.956);
assert(gpas.contains(""));
gpas.erase("Fred");
assert(gpas.size() == 3  &&  gpas.contains("Lucy")  &&  gpas.contains("Ethel")  &&
            gpas.contains(""));
```

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

```
Map m1;
m1.insert("Fred", 2.956);
Map m2;
m2.insert("Ethel", 3.538);
m2.insert("Lucy", 2.956);
m1.swap(m2);
assert(m1.size() == 2  &&  m1.contains("Ethel")  &&  m1.contains("Lucy")  &&
       m2.size() == 1  &&  m2.contains("Fred"));
```

When comparing keys for `insert`, `update`, `insertOrUpdate`, `erase`, `contains`, and the two-argument form of `get`, 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 Map class should be const member functions (because they do not modify the Map), and change the class declaration accordingly.

2. As defined above, the Map class allows the client to use a map that contains only `std::string`s as keys and `double`s as values. Someone who wanted to modify the class to contain keys or values of another type, such as keys being `int`s and values being `int`s, would have to make changes in many places. Modify the class definition you produced in the previous problem to use a type alias for all keys wherever the original definition used a `std::string` for that purpose, and a type alias for all values wherever the original definition used a `double` for that purpose. A *type alias* is a name that is a synonym for some type; here is an example:

   ```
   // The following line introduces the type alias Number as a synonym
   // for the type int; anywhere the code uses the name Number, it means
   // the type int.

using Number = int;

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

   The advantage of using the type alias Number is that if we later wish 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 using statement introducing the type alias Number.

   (Aside: Prior to C++11 (and still usable now), the only way to introduce a type alias was to use a `typedef` statement, e.g. `typedef int Number;`. Appendix A.1.8 of the textbook describes `typedef`.)

   To make the grader's life easier, we'll require that everyone use the same synonyms for their type aliases: You must use the name `KeyType` for the name of the type used for keys, and `ValueType` for the name of the type used for values, with exactly those spellings and cases.

3. Now that you have defined an interface for a map class where the key and the value types can be easily changed, implement the class and all its member functions in such a way that the key/value pairs in a map 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 map must be able to hold a maximum of `DEFAULT_MAX_ITEMS` distinct keys, where

   ```
const int DEFAULT_MAX_ITEMS = 240;
   ```

   (Hint: Define a structure type containing a member of type KeyType and a member of type ValueType. Have Map's array data member be an array of these structures.)

Test your class for a Map that maps `std::string`s to `double`s. Place your class definition, non-inline function prototypes, and inline function implementations (if any) in a file named `Map.h`, and your non-inline function implementations (if any) in a file named `Map.cpp`. (If we haven't yet discussed inline, then if you haven't encountered the topic yourself, all your functions will be non-inline, which is fine.)

Except to add a `dump` function (described below), you must not add public data or function members to, delete functions from, or change the public interface of the Map class. You must not declare any additional struct/class outside the Map class, and you must not declare any *public* struct/class inside the Map class. You may add whatever private data members and private member functions you like, and you may declare *private* structs/classes inside the Map class if you like.

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 implementation of the Map class must be such that the compiler-generated destructor, copy constructor, and assignment operator do the right things. Write a test program named `testMap.cpp` to make sure your Map class implementation works properly. Here is one possible (incomplete) test program:

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

int main()
{
    Map m;  // maps strings to doubles
    assert(m.empty());
    ValueType v = -1234.5;
    assert( !m.get("abc", v)  &&  v == -1234.5); // v unchanged by get failure
    m.insert("xyz", 9876.5);
    assert(m.size() == 1);
    KeyType k = "hello";
    assert(m.get(0, k, v)  &&  k == "xyz"  &&  v == 9876.5);
    cout << "Passed all tests" << endl;
}
```

Now change (only) the two type aliases in `Map.h` so that the Map type will now map `int`s to `std::string`s. Make no other changes to `Map.h`, and make no changes to `Map.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 "Map.h"
#include <iostream>
#include <cassert>
using namespace std;

int main()
{
    Map m;  // maps ints to strings
    assert(m.empty());
    ValueType v = "Ouch";
    assert( !m.get(42, v)  &&  v == "Ouch"); // v unchanged by get failure
    m.insert(123456789, "Wow!");
    assert(m.size() == 1);
    KeyType k = 9876543;
    assert(m.get(0, k, v)  &&  k == 123456789  &&  v == "Wow!");
    cout << "Passed all tests" << endl;
}
```

You may need to flip back and forth a few times to fix your `Map.h` and `Map.cpp` code so that the *only* change to those files you'd need to make to change a map's key and value types is to the two type aliases in `Map.h`. (When you turn in the project, have the type aliases in `Map.h` specify the key type to be `std::string` and the value type to be `double`.)

Except in a using statement in `Map.h` introducing a type alias, the word `double` must not appear in `Map.h` or `Map.cpp`. Except in a using statement introducing a type alias and in the context of `#include <string>` in `Map.h`, the word `string` must not appear in `Map.h` or `Map.cpp`.

(Implementation note 1: If you declare another structure to help you implement a Map, put its declaration in `Map.h` (and `newMap.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 Map class. In fact, to enforce clients' not using that structure type, don't declare it outside of the Map class; instead, declare that helper structure in the *private* section of Map. 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 `Map.cpp` (and `newMap.cpp` for Problem 5).)

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

4. Now that you've implemented the class, write some client code that uses it. We might want a class that keeps track of a fleet of cars and how many miles each car has been driven. Implement the following class:

```
#include "Map.h"

class CarMap
{
  public:
    CarMap();       // Create an empty car map.

    bool addCar(std::string license);
      // If a car with the given license plate is not currently in the map,
      // and there is room in the map, add an entry for that car recording
      // that it has been driven 0 miles, and return true.  Otherwise,
      // make no change to the map and return false.

    double miles(std::string license) const;
      // If a car with the given license plate is in the map, return how
      // many miles it has been driven; otherwise, return -1.

    bool drive(std::string license, double distance);
      // If no car with the given license plate is in the map or if
      // distance is negative, make no change to the map and return
      // false.  Otherwise, increase by the distance parameter the number
      // of miles the indicated car has been driven and return true.

    int fleetSize() const;  // Return the number of cars in the CarMap.

    void print() const;
      // Write to cout one line for every car in the map.  Each line
      // consists of the car's license plate, followed by one space,
      // followed by the number of miles that car has been driven.  Write
      // no other text.  The lines need not be in any particular order.

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

Your CarMap implementation must employ a data member of type Map that uses the type aliases `KeyType` and `ValueType` as synonyms for `std::string` and `double`, respectively. (Notice we said a member of type *Map*, not of type *pointer to Map*.) Except for the using statements introducing the type aliases, you must not make any changes to the `Map.h` and `Map.cpp` files you produced for Problem 3, so you must not add any member functions or data members to the Map class. Each of the member functions `addCar`, `miles`, `drive`, `fleetSize`,

and `print` must delegate as much of the work that they need to do as they can to Map member functions. (In other words, they must not do work themselves that they can have Map member functions do instead.) If the compiler-generated destructor, copy constructor, and assignment operator for CarMap don't do the right thing, declare and implement them. Write a program to test your CarMap class. Name your files `CarMap.h`, `CarMap.cpp`, and `testCarMap.cpp`.

The words `for` and `while` must not appear in `CarMap.h` or `CarMap.cpp`, except in the implementation of `CarMap::print` if you wish. The characters `[` (open square bracket) and `*` must not appear in `CarMap.h` or `CarMap.cpp`, except in comments if you wish. You do not have to change `std::string` to `KeyType` and `double` to `ValueType` in `CarMap.h` and `CarMap.cpp` if you don't want to (since unlike Map, which is designed for a wide variety of key and value types, CarMap is specifically designed to work with strings and doubles). In the code you turn in, `CarMap`'s member functions must not call `Map::dump`.

5. Now that you've created a map 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 `newMap.h`, `newMap.cpp`, and `testnewMap.cpp`. Update those files by either adding another constructor or modifying your existing constructor so that a client can do the following:

```
Map a(1000);   // a can hold at most 1000 key/value pairs
Map b(5);      // b can hold at most 5 key/value pairs
Map c;         // c can hold at most DEFAULT_MAX_ITEMS key/value pairs
KeyType k[6] = { a list of six distinct values of the appropriate type };
ValueType v  = a value of the appropriate type;

  // No failures inserting pairs with 5 distinct keys into b
for (int n = 0; n < 5; n++)
    assert(b.insert(k[n], v));

  // Failure if we try to insert a pair with a sixth distinct key into b
assert(!b.insert(k[5], v));

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

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, and you may add or remove private members.) Change the implementation of the `swap` function so that the number of statement executions when swapping two maps is the same no matter how many key/value pairs are in the maps. (You would not satisfy this requirement if, for example, your swap function caused a loop to visit each pair in the map, since the number of statements executed by all the iterations of the loop would depend on the number of pairs in the map.)

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

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

Verify that your CarMap class still works properly with this new version of Map (with `KeyType` and `ValueType` being type aliases for `std::string` and `double`, respectively). You should not need to change your CarMap class or its implementation in any way, other than to include `"newMap.h"` instead of `"Map.h"`. (For this test, be sure to link with `newMap.cpp`, not `Map.cpp`.) (Before you turn in `CarMap.h` and `CarMap.cpp`, be sure to restore any #includes to `"Map.h"` instead of `"newMap.h"`.)

Turn it in

By Monday, January 20, 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 `Map.h`, `CarMap.h`, and `newMap.h` must have an appropriate include guard. In the files you turn in, the using statements in `Map.h` and `newMap.h` must introduce `KeyType` as a type alias for `std::string` and `ValueType` as a type alias for `double`.

- If we create a project consisting of `Map.h`, `Map.cpp`, and `testMap.cpp`, it must build successfully under both g32 and either Visual C++ or clang++. (Note: To build an executable using g32 from some, but not all, of the .cpp files in a directory, you list the .cpp files to use in the command. To build an executable named `req1` for this requirement, for example, you'd say `g32 -o req1 Map.cpp testMap.cpp`.)

- If we create a project consisting of `Map.h`, `Map.cpp`, `CarMap.h`, `CarMap.cpp`, and `testCarMap.cpp`, it must build successfully under both g32 and either Visual C++ or clang++.

- If we create a project consisting of `newMap.h`, `newMap.cpp`, and `testnewMap.cpp`, it must build successfully under both g32 and either Visual C++ or clang++.

- If we create a project consisting of `newMap.h`, `newMap.cpp`, and `testMap.cpp`, where in `testMap.cpp` we change only the `#include "Map.h"` to `#include "newMap.h"`, the project must build successfully under both g32 and either Visual C++ or clang++. (If you try this, be sure to change the `#include` back to `"Map.h"` before you turn in `testMap.cpp`.)

- The source files you submit for this homework must not contain the word `friend` or `pragma` 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, `CarMap::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 Map and CarMap. 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 `Map::erase` or `Map::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:

```
bool Map::erase(const KeyType& value)
{
    return true;  // not correct, but at least this compiles
}

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

- Given `Map.h` with the type alias for the Map's key type specifying `std::string` and the type alias for its value type specifying `double`, if we make no change to your `Map.cpp`, then if we compile your `Map.cpp` and link it to a file containing

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

```
using namespace std;

void test()
{
    Map m;
    assert(m.insert("Fred", 2.956));
    assert(m.insert("Ethel", 3.538));
    assert(m.size() == 2);
    ValueType v = 42;
    assert(!m.get("Lucy", v)  &&  v == 42);
    assert(m.get("Fred", v)  &&  v == 2.956);
    v = 42;
    KeyType x = "Lucy";
    assert(m.get(0, x, v)  &&
           ((x == "Fred"  &&  v == 2.956)  ||  (x == "Ethel"  &&  v == 3.538)));
    KeyType x2 = "Ricky";
    assert(m.get(1, x2, v)  &&
           ((x2 == "Fred"  &&  v == 2.956)  ||  (x2 == "Ethel"  &&  v == 3.538))  &&
           x != x2);
}

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 `Map.h` change the Map's type aliases to specify `int` as the key type and `std::string` as the value type without making any other changes, recompile `Map.cpp`, and link it to a file containing

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

void test()
{
    Map m;
    assert(m.insert(10, "diez"));
    assert(m.insert(20, "veinte"));
    assert(m.size() == 2);
    ValueType v = "cuarenta y dos";
    assert(!m.get(30, v)  &&  v == "cuarenta y dos");
    assert(m.get(10, v)  &&  v == "diez");
    v = "cuarenta y dos";
    KeyType x = 30;
    assert(m.get(0, x, v)  &&
           ((x == 10  &&  v == "diez")  ||  (x == 20  &&  v == "veinte")));
    KeyType x2 = 40;
    assert(m.get(1, x2, v)  &&
           ((x2 == 10  &&  v == "diez")  ||  (x2 == 20  &&  v == "veinte"))  &&
           x != x2);
}

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 `newMap.h` with the type alias for the Map's key type specifying `std::string` and the type alias for its value type specifying `double`, if we make no change to your `newMap.cpp`, then if we compile your `newMap.cpp` and link it to a file containing

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

void test()
{
    Map m;
    assert(m.insert("Fred", 2.956));
    assert(m.insert("Ethel", 3.538));
    assert(m.size() == 2);
    ValueType v = 42;
    assert(!m.get("Lucy", v)  &&  v == 42);
    assert(m.get("Fred", v)  &&  v == 2.956);
    v = 42;
    KeyType x = "Lucy";
    assert(m.get(0, x, v)  &&
           ((x == "Fred"  &&  v == 2.956)  ||  (x == "Ethel"  &&  v == 3.538)));
    KeyType x2 = "Ricky";
    assert(m.get(1, x2, v)  &&
           ((x2 == "Fred"  &&  v == 2.956)  ||  (x2 == "Ethel"  &&  v == 3.538))  &&
           x != x2);
}

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 `newMap.h` change the Map's type aliases to specify `int` as the key type and `std::string` as the value type without making any other changes, recompile `newMap.cpp`, and link it to a file containing

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

void test()
{
    Map m;
    assert(m.insert(10, "diez"));
    assert(m.insert(20, "veinte"));
    assert(m.size() == 2);
    ValueType v = "cuarenta y dos";
    assert(!m.get(30, v)  &&  v == "cuarenta y dos");
    assert(m.get(10, v)  &&  v == "diez");
    v = "cuarenta y dos";
    KeyType x = 30;
    assert(m.get(0, x, v)  &&
           ((x == 10  &&  v == "diez")  ||  (x == 20  &&  v == "veinte")));
    KeyType x2 = 40;
    assert(m.get(1, x2, v)  &&
           ((x2 == 10  &&  v == "diez")  ||  (x2 == 20  &&  v == "veinte"))  &&
           x != x2);
}
```

```
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 `testMap.cpp`, `testCarMap.cpp`, and `testnewMap.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.