In this project, you will write the implementation of the Map type to use a data structure of your choice. You must *not* use arrays. You will also implement a couple of algorithms that operate on maps.

Implement Map

Consider the following Map interface:

```
using KeyType = std::string;
using ValueType = double;

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

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

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

    bool insert(const KeyType& key, const ValueType& 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).

    bool update(const KeyType& key, const ValueType& 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 KeyType& key, const ValueType& 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.
      // If key is not equal to any key currently in the map then add it and
      // return true. In fact this function always returns true.

    bool erase(const KeyType& 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 KeyType& key) const;
      // Return true if key is equal to a key currently in the map, otherwise
      // false.

    bool get(const KeyType& key, ValueType& value) const;
      // If key is equal to a key currently in the map, set value to the
      // value in the map that 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, KeyType& key, ValueType& value) const;
      // 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.
};
```

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 all 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 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 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.

For this project, implement this Map interface using your choice of data structure (dynamically resizeable array, singly-linked list, doubly-linked list, tree, or hash table). (You must not use any contaner from the C++ library.)

For your implementation, if you let the compiler write the destructor, copy constructor, and assignment operator, they will do the wrong thing, so you will have to declare and implement these public member functions as well:

Destructor
When a Map is destroyed, all dynamic memory must be deallocated.
Copy constructor
When a brand new Map is created as a copy of an existing Map, a deep copy should be made.
Assignment operator
When an existing Map (the left-hand side) is assigned the value of another Map (the right-hand side), the result must be that the left-hand side object is a duplicate of the right-hand side object, with no memory leak (i.e. no memory from the old value of the left-hand side should be still allocated yet inaccessible).

Notice that there is now no *a priori* limit on the maximum number of key/value pairs in the Map (so `insertOrUpdate` should always return true). Also, if a Map has a size of *n*, then the values of the first parameter to the three-parameter form of `get` for which that function retrieves a key and a value and returns true are *0*, *1*, *2*, ..., *n-1*; for other values, it returns false without setting its second and third parameters.

Another requirement is that the number of statement executions when swapping two maps must be the same no matter how many key/value pairs are in the maps.

Implement some map algorithms

Using only the *public* interface of Map, implement the following two functions. (Notice that they are *non-member* functions; they are *not* members of Map or any other class.)

`bool combine(const Map& m1, const Map& m2, Map& result);`

When this function returns, `result` must consist of pairs determined by these rules:

- If a key appears in exactly one of `m1` and `m2`, then `result` must contain a pair consisting of that key and its corresponding value.
- If a key appears in both `m1` and `m2`, with the same corresponding value in both, then `result` must contain a pair with that key and value.

When this function returns, `result` must contain no pairs other than those required by these rules. (You must *not* assume `result` is empty when it is passed in to this function; it might not be.)

If there exists a key that appears in both `m1` and `m2`, but with different corresponding values, then this function returns false; if there is no key like this, the function returns true. Even if the function returns false, `result` must be constituted as defined by the above rules.

For example, suppose a Map maps strings to doubles. If `m1` consists of the three pairs (in any order)

```
    "Fred"  123      "Ethel"  456      "Lucy"  789
```

and `m2` consists of (in any order)

```
    "Lucy"  789      "Ricky"  321
```

then no matter what value it had before, `result` must end up as a map consisting of (in any order)

```
    "Fred"  123      "Ricky"  321      "Lucy"  789     "Ethel"  456
```

and `combine` must return true. If instead, `m1` were as before, and `m2` consisted of

```
    "Lucy"  654      "Ricky"  321
```

then no matter what value it had before, `result` must end up as a map consisting of (in any order)

```
    "Fred"  123      "Ricky"  321      "Ethel"  456
```

and `combine` must return false.

`void subtract(const Map& m1, const Map& m2, Map& result);`

When this function returns, `result` must contain a copy of all the pairs in `m1` whose keys don't appear in `m2`; it must not contain any other pairs. (You must *not* assume `result` is empty when it is passed in to this function; it may not be.)

For example, if `m1` consists of the three pairs (in any order)

```
    "Fred"  123      "Ethel"  456      "Lucy"  789
```

and `m2` consists of (in any order)

```
    "Lucy"  789      "Ricky"  321      "Ethel"  654
```

then no matter what value it had before, `result` must end up as a map consisting of

```
    "Fred"  123
```

If English is not your native language, make extra sure you spell the name of this function correctly: it's `subtract`, not `substract`.

Be sure these functions behave correctly in the face of *aliasing*: What if `m1` and `result` refer to the same Map, for example?

Other Requirements

Regardless of how much work you put into the assignment, your program will receive a low score for correctness if you violate these requirements:

- Your class definition, declarations for the two required non-member functions, and the implementations of any functions you choose to inline must be in a file named Map.h, which must have appropriate include guards. The implementations of the functions you declared in Map.h that you did not inline must be in a file named Map.cpp. Neither of those files may have a main routine (unless it's commented out). You may use a separate file for the main routine to test your Map class; you won't turn in t

- Except to add a destructor, copy constructor, assignment operator, and `dump` function (described below), you must not add functions 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. The source files you submit for this project must not contain the word `friend`. You must not use any global variables whose values may be changed during execution.

  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`.

- `Map.cpp` must not contain the word `string` or `double` except for `Convert_Key` functions. (`Map.h` may contain them only in the typedef statements, and must contain `#include <string>` if a typedef statement contains the word `string`.)

- Your code must build successfully (under both Visual C++ and either clang++ or g++) if linked with a file that contains a main routine.

- You must have an implementation for every member function of Map, as well as the non-member functions `combine` and `subtract`. Even if you can't get a function implemented correctly, it must have an implementation that at least builds successfully. For example, if you don't have time to correctly implement `Map::erase` or `subtract`, say, here are implementations that meet this requirement in that they at least build successfully:

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

        void subtract(const Map& m1, const Map& m2, Map& result)
        {
            // does nothing; not correct, but at least this compiles
        }
```

  You've probably met this requirement if the following file compiles and links with your code. (This uses magic beyond the scope of CS 30.)

```
        #include "Map.h"
        #include <type_traits>

        #define CHECKTYPE(f, t) { auto p = (t)(f); (void)p; }

        static_assert(std::is_default_constructible<Map>::value,
                "Map must be default-constructible.");
        static_assert(std::is_copy_constructible<Map>::value,
                "Map must be copy-constructible.");

        void ThisFunctionWillNeverBeCalled()
        {
            CHECKTYPE(&Map::operator=,      Map& (Map::*)(const Map&));
            CHECKTYPE(&Map::empty,          bool (Map::*)() const);
            CHECKTYPE(&Map::size,           int  (Map::*)() const);
            CHECKTYPE(&Map::insert,         bool (Map::*)(const KeyType&, const ValueTyp
e&));
            CHECKTYPE(&Map::update,         bool (Map::*)(const KeyType&, const ValueTyp
e&));
            CHECKTYPE(&Map::insertOrUpdate, bool (Map::*)(const KeyType&, const ValueTyp
e&));
            CHECKTYPE(&Map::erase,          bool (Map::*)(const KeyType&));
            CHECKTYPE(&Map::contains,       bool (Map::*)(const KeyType&) const);
            CHECKTYPE(&Map::get,            bool (Map::*)(const KeyType&, ValueType&) cons
t);
            CHECKTYPE(&Map::get,            bool (Map::*)(int, KeyType&, ValueType&) const);
            CHECKTYPE(&Map::swap,           void (Map::*)(Map&));

            CHECKTYPE(combine,  bool (*)(const Map&, const Map&, Map&));
            CHECKTYPE(subtract, void (*)(const Map&, const Map&, Map&));
        }

        int main()
        {}
```

- If you add `#include <string>` to `Map.h`, have Map's typedefs define `KeyType` as `std::string` and `ValueType` as `double`, and link your code to a file containing

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

        void test()
        {
            Map m;
            assert(m.insert("Fred", 123));
            assert(m.insert("Ethel", 456));
            assert(m.size() == 2);
            double d = 42;
            assert(m.get("Fred", d)  &&  d == 123);
            d = 42;
            string s1;
            assert(m.get(0, s1, d)  &&
                   ((s1 == "Fred"  &&  d == 123)  ||  (s1 == "Ethel"  &&  d == 456)));
            string s2;
            assert(m.get(1, s2, d)  &&  s1 != s2  &&
                   ((s2 == "Fred"  &&  d == 123)  ||  (s2 == "Ethel"  &&  d == 456)));
        }

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

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

- If we successfully do the above, then make no changes to `Map.h` other than to change the typedefs for Map so that `KeyType` specifies `int` and `ValueType` specifies `std::string`, 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(123, "Fred"));
            assert(m.insert(456, "Ethel"));
            assert(m.size() == 2);
            string s;
            assert(m.get(123, s)  &&  s == "Fred");
            s = "";
            int i1;
            assert(m.get(0, i1, s)  &&
                   ((i1 == 123  &&  s == "Fred")  ||  (i1 == 456  &&  s == "Ethel")));
            int i2;
            assert(m.get(1, i2, s)  &&  i1 != i2  &&
                   ((i2 == 123  &&  s == "Fred")  ||  (i2 == 456  &&  s == "Ethel")));
        }

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

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

- During execution, if a client performs actions whose behavior is defined by this spec, your program must not perform any undefined actions, such as dereferencing a null or uninitialized pointer.

- Your code in `Map.h` and `Map.cpp` must not read anything from `cin` and must not write anything whatsoever to `cout`. If you want to print things out for debugging purposes, write to `cerr` instead of `cout`. `cerr` is the standard error destination; items written to it by default go to the screen. When we test your program, we will cause everything written to `cerr` to be discarded instead ï¿½ we will never see that output, so you may leave those debugging output statements in your program if you wish.

- If you decide to implement the Map using a hash table you must also declare and define the following two non member functions in `Map.cpp`. If a KeyType is specified by a string it will call the first function, if it is specified by a double it should call the second function. You will then mod the returned number by the size of your table.

```
        int Convert_Key(string key)
        {
            // your code for hashing a string goes here
        }

        int Convert_Key(double key)
        {
            // your code for hashing a double goes here
        }

        // below is how you would use the function with the size of your table
        Convert_Key( key ) % TABLESIZE
```