# Programming Assignment 2
# Double Trouble

**Time due: 9:00 PM Tuesday, January 31**

Homework 1 gave you extensive experience with the Map ADT using both arrays and dynamically-allocated arrays. In this project, you will re-write the implementation of your Map type to employ a doubly-linked list rather than an array. You must *not* use arrays. You will also implement a couple of algorithms that operate on maps.

**Implement Map yet again**

Consider the Map interface from problem 2 of Homework 1:

```
typedef TheTypeOfTheKeysGoesHere KeyType;
typedef TheTypeOfTheValuesGoesHere ValueType;

class Map
{
  public:
    Map();
    bool empty() const;
    int size() const;
    bool insert(const KeyType& key, const ValueType& value);
    bool update(const KeyType& key, const ValueType& value);
    bool insertOrUpdate(const KeyType& key, const ValueType& value);
    bool erase(const KeyType& key);
    bool contains(const KeyType& key) const;
    bool get(const KeyType& key, ValueType& value) const;
    bool get(int i, KeyType& key, ValueType& value) const;
    void swap(Map& other);
};
```

In problem 3 of Homework 1, you implemented this interface using an array. For this project, implement the Map interface using a doubly-linked list. (You must not use the `list` class template from the C++ library.)

For the array implementation of problem 3 of Homework 1, since you declared no destructor, copy constructor, or assignment operator, the compiler wrote them for you, and they did the right thing. For this linked list 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, the nodes in the linked list must be deallocated.

Copy constructor
: When a brand new Map is created as a copy of an existing Map, enough new nodes must be allocated to hold a duplicate of the original list.

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 of list nodes (i.e. no list node 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 items in the Map.

Another requirement is that as in Problem 5 of Homework 1, the number of statement executions when swapping two maps must be the same no matter how many items 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 theese 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` had the three pairs (in any order)

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

and `m2` contained (in any order)

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

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

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

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

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

then no matter what value it had before, `result` must end up as a map containing (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` had the three pairs (in any order)

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

and `m2` contained (in any order)

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

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

```
"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 zero 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 that separate file.
- 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` or the character `[` (open square bracket).

  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`.

- You must have an implementation for every member function of Map, as well as the non-member functions `combine` and `subtract`. If you can't get a function implemented correctly, it must at least compile 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 compile 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 32.)

```
#include "Map.h"

void thisFunctionWillNeverBeCalled()
{
    Map m;
    Map m2(m);
    m2 = m;
    bool (Map::*p1)() const                              = &Map::empty;
    int  (Map::*p2)() const                              = &Map::size;
    bool (Map::*p3)(const KeyType&, const ValueType&) = &Map::insert;
    bool (Map::*p4)(const KeyType&, const ValueType&) = &Map::update;
    bool (Map::*p5)(const KeyType&, const ValueType&) = &Map::insertOrUpdate;
    bool (Map::*p6)(const KeyType&)                      = &Map::erase;
    bool (Map::*p7)(const KeyType&) const                = &Map::contains;
    bool (Map::*p8)(const KeyType&, ValueType&) const = &Map::get;
    bool (Map::*p9)(int, KeyType&, ValueType&) const  = &Map::get;
    void (Map::*p10)(Map&)                               = &Map::swap;

    bool (*p11)(const Map&, const Map&, Map&)         = combine;
    void (*p12)(const Map&, const Map&, Map&)         = subtract;
}

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);
    doubld 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 more to `cout`.

- If we successfully do the above, then make no changes to `Map.h` other that 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 more 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 — we will never see that output, so you may leave those debugging output statements in your program if you wish.

**Turn it in**

By Monday, January 30, there will be a link on the class webpage that will enable you to turn in your source files and report. You will turn in a zip file containing these three files:

- `Map.h`. When you turn in this file, the typedefs must specify `std::string` as the KeyType and `double` as the ValueType.
- `Map.cpp`. Function implementations should be appropriately commented to guide a reader of the code.
- `report.doc` or `report.docx` (in Microsoft Word format) or `report.txt` (an ordinary text file) that contains:
  - a description of the design of your doubly-linked list implementation. (A couple of sentences will probably suffice, perhaps with a picture of a typical Map and an empty Map. Is the list circular? Does it have a dummy node? What's in your list nodes? Are they in any particular order?)
  - pseudocode for non-trivial algorithms (e.g., `Map::erase` and `subtract`).
  - a list of test cases that would thoroughly test the functions. Be sure to indicate the purpose of the tests. For example, here's the beginning of a presentation in the form of code:

    The tests were performed on a map from strings to doubles

    ```
      // default constructor
    Map m;
      // For an empty map:
    assert(m.size() == 0);      // test size
    assert(m.empty());          // test empty
    assert(!m.erase("Ricky"));  // nothing to erase
    ```

    Even if you do not correctly implement all the functions, you must still list test cases that would test them. Don't lose points by thinking "Well, I didn't implement this function, so I won't bother saying how I would have tested it if I *had* implemented it."