Winter 2020 CS 32

Programming Assignment 2: Double Trouble

Time due: 11:00 PM Tuesday, January 28

Homework 1 gave you extensive experience with the Map type using both arrays and dynamically-allocated arrays. In this project, you will re-write the implementation of the 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:

```
using KeyType = TheTypeOfTheKeysGoesHere;
using ValueType = TheTypeOfTheValuesGoesHere;

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 this 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 key/value pairs in the Map (so `insertOrUpdate` should always return true). Notice also that, as in Homework 1, 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. This is the same visible behavior as in Homework 1.

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 key/value pairs are in the maps.

Implement some map algorithms

Implement the following two functions. Notice that they are *non-member* functions: They are *not* members of Map or any other class, so they must *not* access *private* members of Map.

```
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 exactly one 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 you like)

```
"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 you like)

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

and `combine` must return false.

```
void reassign(const Map& m, Map& result);
```

Imagine a dance with two groups of people, K and V, with the same number of people in each group. During a dance, each person in group K is dancing with a person in group V. At a given signal, they change partners so that each person in group K is now dancing with a person in group V different from the person they were dancing with before.

When the `reassign` function returns, `result` must contain, for each pair p_1 in `m`, a pair with p_1's key mapping to a value copied from a different pair p_2 in `m`, and no other pair in `result` has its value copied from p_2. (At the dance, if k_1's original partner v_1 is replaced by k_2's original partner v_2, then no person in group K other than k_1 also ends up dancing with v_2.) However, if `m` has only one pair, then `result` must contain simply a copy of that pair. (You can't change partners if you're the only couple dancing!)

Upon return, `result` must contain the same number of pairs as `m`; you must *not* assume `result` is empty when it is passed in to this function; it may not be.

For example, if `m` consists of the four pairs (in any order)

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

then no matter what value it had before, `result` must end up as a map consisting of one of the following groups of four pairs (with the pairs in `result` being in any order you like):

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

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

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

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

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

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

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

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

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

but not, say, one of these:

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

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

(In the first, Lucy didn't change partners; in the the second, both Ethel and Lucy ended up with Ricky's original partner.)

As another example, if `m` consists of the three pairs (in any order)

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

then no matter what value it had before, `result` must end up as a map consisting of one of the following groups of three pairs (with the pairs in `result` being in any order you like):

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

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

If the result were the first, Fred must have ended up with Lucy's 456 and Lucy ended up with Ethel's 456. In the second case, Fred must have ended up with Ethel's 456, and Ethel ended up with Lucy's 456.

Notice that this spec does not require any particular one of the possible reassignments and does not require that the reassignment be randomly chosen. (Hint: This function can thus be implemented without its

having to repeatedly examine an auxiliary array or other container that holds a collection of many items.)

Be sure that in the face of *aliasing*, these functions behave as this spec requires: Does your implementation work correctly 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 `pragma` or the character `[` (open square bracket). You must not use any global variables whose values may be changed during execution. (Global *constants* are fine.)

- 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.)

  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`. (`Map.h` may contain them only in using statements introducing type aliases, and must contain `#include <string>` if a using statement introducing a type alias contains the word `string`.)

- Your code must build successfully (under both g32 and either Visual C++ or clang++) 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 `reassign`. 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 `reassign`, 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 reassign(const Map& m, 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"
#include <type_traits>

#define CHECKTYPE(f, t) { auto p = static_cast<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.");
static_assert(std::is_copy_assignable<Map>::value,
        "Map must be assignable.");

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 ValueType&));
    CHECKTYPE(&Map::update,         bool (Map::*)(const KeyType&, const ValueType&));
    CHECKTYPE(&Map::insertOrUpdate, bool (Map::*)(const KeyType&, const ValueType&));
    CHECKTYPE(&Map::erase,          bool (Map::*)(const KeyType&));
    CHECKTYPE(&Map::contains,       bool (Map::*)(const KeyType&) const);
    CHECKTYPE(&Map::get,            bool (Map::*)(const KeyType&, ValueType&) const);
    CHECKTYPE(&Map::get,            bool (Map::*)(int, KeyType&, ValueType&) const);
    CHECKTYPE(&Map::swap,           void (Map::*)(Map&));

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

int main()
{}
```

- If you add `#include <string>` to `Map.h`, have the type alias for Map's key type specify `std::string`, and have the type alias for its value type specify `double`, then if we make no change to your `Map.cpp`, compile it, and link it to a file containing

```
#include "Map.h"
#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);
    ValueType v = 42;
    assert(!m.get("Lucy", v)  &&  v == 42);
    assert(m.get("Fred", v)  &&  v == 123);
    v = 42;
    KeyType x = "Lucy";
    assert(m.get(0, x, v)  &&
           ((x == "Fred"  &&  v == 123)  ||  (x == "Ethel"  &&  v == 456)));
    KeyType x2 = "Ricky";
    assert(m.get(1, x2, v)  &&
           ((x2 == "Fred"  &&  v == 123)  ||  (x2 == "Ethel"  &&  v == 456))  &&
```

```
            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` to `cout`and nothing else to `cout`, and terminate normally.

- If we successfully do the above, then make no changes to `Map.h` other than to change the type aliases for Map so that `KeyType` specifies `int` and `ValueType` specifies `std::string`, make no changes to `Map.cpp`, recompile `Map.cpp`, and link it to a file containing

```
#include "Map.h"
#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` to `cout`and nothing else to `cout`, and terminate normally.

- 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 27, 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 using statements 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.docx` or `report.doc` (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 `reassign`).
  - 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."