Assignment 2: STL HashMap

Assignment due on Paperless on Wednesday, June 7th at 11:59 PM (FIRM DEADLINE)

Introduction and Assignment Goals

According to veteran C++ programmers, there are two projects which bring together all the knowledge that a proficient C++ programmer should have:

1. Implementing a STL-compliant template class; and
2. Implementing a macro to hash string literals at compile-time.

In this assignment, we will be putting **#1** on your resume by completing a STL-compliant **HashMap**.

Recall that the `map` abstract data type stores key -- value pairs, and provides efficient lookup of the value of any key. There are two versions of this in the STL: `std::map`, traditionally implemented using a balanced binary search tree, and `std::unordered_map` (introduced in C++11), which uses a clever technique called hashing and supports the most map operations in constant time (!).

In this assignment, you will be given a nearly complete implementation of a HashMap (the name of this data structure in the Stanford C++ library). The starter code is nearly complete, but is not "STL-compliant"; for example, copy and move semantics are not supported yet. Your goal is to extend it so it becomes an STL-compliant, industrial strength, robust, and blazingly fast data structure.

A hashmap is a bit of a complicated data structure, but we've given you nearly all of the implementation of it. In brief, this is how it works:

- A hashmap stores an array (represented by the private vector `_buckets_array` in our HashMap) of "buckets".
- Each "bucket" is a linked list: linked lists are made up of `node`s, which are defined in `hashmap.h`.
- Each `node` in these linked lists contains one key-value pair that is contained in our map. In the `node` struct, this is the field `value`, of type `value_type`. `value_type` is a **type-alias** for a `std::pair<K, M>`. Check out its definition near the top of the .h file!
- When a user of the hashmap class tries to look up or insert a key, the HashMap class first converts the key, no matter what type it is, to an integer. It does this using the hash function, stored in the private member variable `_hash_function`. For our purposes, this will always be the default hash function in the STL. Then, the HashMap class mods that integer by the number of buckets in `_buckets_array`. This is the index at which HashMap will insert or look for the key. From here, its just a matter of traversing the linked list stored at that index to find the key in question (it is possible that multiple keys are stored in the same index because their hashes, or the mod of their hashes, are the same!)
- The HashMap will occasionally increase the number of buckets and re-hash and store everything to make sure each index doesn't have too many keys stored there. This keeps all operations very fast, because it is very quick to look up an index in an array, and we are guaranteed to have very short linked lists there!

Dont worry if you don't feel super comfortable with HashMaps! We have implemented all of this logic for you, and we highly encourage you to read through the code to understand what is happening. However, this assignment does not ask you to implement anything except some SMFs, which only deal with initializing the private member variables.

You will not necessarily write a lot of code on this assignment (~20 Lines), but some of your time will be spent on understanding the starter code given to you, and reasoning about design choices and common C++ idioms. **You may optionally work with a partner. We recommend having a partner to discuss the design questions.** If you choose to work with a partner, we ask that you and your partner submit a single version of the assignment among both of you. This means that you will ideally be working with your partner on a shared version of the code and short answer questions. Feel free to start a shared Google Doc, a private GitHub repository, or anything else that you'd need to keep track of a shared version of the code. You may also choose to schedule video chat sessions with each other to work on the assignment.

Download And Set Up Assignment

1. We suggest running the code on the Myth machines to prevent any issues with your computer's setup. If you haven't already, set yourself up in VS Code. The instructions for this are on the assignment setup page. We recommend the alternate SSH setup.
2. Download the starter code here: Starter Code
3. In terminal, `scp` your Assignment 3 starter code to your 106L folder in myth. `scp LOCAL_PATH <yoursunetid>@myth.stanford.edu:MYTH_PATH`
   LOCAL_PATH is the place on your laptop where you just downloaded the starter code. It will probably be something like `Downloads/HashMap_Starter.zip`.
   MYTH_PATH is the path on myth where you keep your CS106L folder.
   You can find this by 1) `ssh`ing into Myth using `ssh <yoursunetid>@myth.stanford.edu`, 2) entering your password and then 3) `cd`ing to your CS106L directory and finally 4) printing out the path to this folder using `pwd`.
4. Unzip the folder in myth by running `unzip HashMap_Starter.zip`.
5. Now, you should be able to open your Assignment 3 code using whichever VS Code setup instructions you have been using all quarter!

Feel free to email the lecturers if you are having trouble with setup!

Starter Code

Please download the starter code here. The starter code contains the following files:

- `hashmap.h`: This file contains the (incomplete) class function prototypes for the HashMap, and also contains thorough comments about each function. **You will need to modify many of these function prototypes for milestone 1, and add a few for milestone 2** Like in CS 106B, you may add any private helper functions (though it is unlikely you will need to), but do not add or change the prototypes of any public functions. Imagine what a disaster it would be for the world if the vector's size function was renamed to length or if the erase function had a few extra parameters...a good chunk of the world's C++ code would stop compiling!
- `hashmap.cpp`: This file contains the implementation of the HashMap, and also contains thorough comments about implementation details. **Besides changing function signatures for milestone 1, you will have to add the implementations for milestone 2 to this file.**
- `hashmap_iterator.h`: This file contains the complete class declaration and implementation for a HashMapIterator, and also contains thorough comments about each function. You will be reading the code here for milestone 1, but you will not need to modify this file at all. **We have never used a class as our iterator before, make sure you read through these files and try to understand what is going on! Importantly, check out the difference between the `iterator` and `const_iterator` aliases in `hashmap.h`**
- `short_answer.txt`: You will write your solutions to the short answer questions in this file.
- `build.sh`: This file contains the compilation instructions for the project. To check if your Hashmap compiles, run `./build.sh` from your assignment 4 directory.
- `main.sh`: This file contains example code using your The hashmap. To run this code, first build it and then run `./main`
- `tests.cpp` and `test_settings.cpp`: These files contain crazy amounts of code meant to test your hashmap! You shouldn't need to edit them!

Tips

You compile and run this code the same way you would lecture code! In the terminal from your `HashMap_Starter` directory, run `./build.sh` to compile and `./main` to run your code!

Make sure the lines marked "UNCOMMENT FOR MILESTONE 2" are commented out for now. You will get tons of compiler errors!

Before changing any functions, your code should compile. After chaning the functions in main, don't freak out about the loads of compile errors! Just keep looking for functions in HashMap to mark const until you get to a reasonable point in the compile messages, and then try to use those to find more functions that need a const or an overload!

Make sure you change your function signatures in two places: the .h **and** the .cpp

The functions in main are declared before theyre implemented, so any signatures you change there should also change where they are first defined as well!

Its easy to make a lot of copy-paste errors here. Remember when you are overloading, you are also changing the return type.

Notice how `begin()` is overloaded with a const version! Even though only the return type is changed, this type of overloading is ok because the function is marked `const`, which means the compiler can figure out which version of `begin()` to use based on whether the HashMap its been called on is const or not!

Milestone 1: Const-Correctness

Your first task will be to make a **const-interface** for the HashMap class. Recall that we often mark variables, especially function parameters `const` to avoid a specific kind of bug; modifying a piece of data when we weren't supposed to. **We highly recommend checking out the end of the Template Classes and Const-Correctness lecture before starting!** We have already marked a couple member functions const for you (check out the const overload of `begin()`, it is up to you to find the rest! Here is what you need to do to complete this milestone:

1. Head to `main.cpp` and pay special attention to all the helper functions called by `student_main()`. None of them have bugs, but some of them don't properly mark their parameters `const` when necessary. Mark the appropriate parameters `const`. Now, if you try to build, you will notice your code doesn't compile! Thats because most of your HashMap functions are not marked `const`.
2. Head to your `hashmap.cpp` and `hashmap.h` files. Go ahead and mark any existing functions `const`, as long as this is appropriate. This is step 1 for creating your const-interface! Remember, any signatures you change `hashmap.h`, you also need to change in `hashmap.cpp`.
3. Finally, you will need to add a few functions (really, you will need to overload existing ones to have a `const` version) to make your const-interface robust. Recall that iterators by default allow us to dereference and change their underlying value, but sometimes we just want to use an iterator to, well, **iterate**, without changing any values. Make use of the `const_iterator` alias we provided you in `hashmap.h` to create const versions of the appropriate functions.
   - **HINT:** The main challenge here is identifying which functions to add. You will have to do little else than call the non-const version of the function in your new functions. As an example, we have given you a const version of `begin()`. Use the `static_cast<const>` trick demonstrated there in the rest of your functions!
   - **HINT:** Look at all the functions which return an iterator. Which of these should be overloaded to return a `const_iterator` when necessary?
   - **HINT:** If you have marked the appropriate parameters `const` in `main.cpp`, any call to a non-const function will error, you can use this as a guide for which functions you might want to overload! There is one function that does not return an iterator that still must be overloaded, can you find it?
4. Finally, uncomment lines 5, 6, and 28 (the ones with the comment "UNCOMMENT FOR MILESTONE 2"), and try to compile again. If you missed any functions that needed to be marked const or overloaded, you will see that calls to those functions error in the compiler error messages!

Writeup

Please answer these questions in `short_answer.txt`

1. at() vs []

   Explain the difference between `at()` and the implementation of the operator `[]`. Why did you have to overload one and not the other?
   Hint: You will likely only need to read the header comments to do this

2. Find vs. *Find*

   In addition to the `HashMap::find` member function, there is also a `std::find` function in the STL algorithms library. If you were searching for key k in HashMap m, is it preferable to call `m.find(k)` or `std::find(m.begin(), m.end(), k)`?
   Hint: on average, there are a constant number of elements per bucket. Also, one of these functions has a faster Big-O complexity because one of them uses a loop while another does something smarter.

3. RAII?

   This HashMap class is RAII-compliant. Explain why.

4. Increments

   Briefly explain the implementation of HashMapIterator's operator++, which we provide for you. How does it work and what checks does it have?

Milestone 2: Special Member Functions and Move Semantics

Any good STL-compliant class must have correct **special member functions.** Recall that there are six major special member functions:

- Default constructor *(implemented for you)*
- Destructor *(implemented for you)*
- Copy constructor
- Copy assignment operator
- Move constructor
- Move assignment operator

The first two are implemented for you; your job is to implement the last four. Specifically, the **copy** operations should create an identical copy of the provided HashMap, while the **move** operations should move the contents of the provided HashMap to `this` (the instance of the HashMap upon which the move operation is called). Avoid memory leaks and perform your copy/move as efficiently and safely as possible.

Hints

- You will need to move/copy the _buckets_array. Here is the one place you may have to work with pointers. One step of the first steps of copy should be to create a vector<node*> that is initialized to *nullptr*s. Then, you have a valid empty HashMap that you can start copying elements over using the public member functions.
- `clear()` and `insert()` change the `_size` member variable, so keep that in mind when changing _size yourself. If you see that size is twice what you expect it to be, then you are probably incrementing _size twice - once in insert(), and once in your code.
- Be careful about self-assignment! **HINT:** Use `clear()` and `insert()` :)
- In the edge case tests you may see chained assignments (e.g. map = map = map). The assignment operator is right-associative, and also returns a reference to the HashMap itself, so this gets interpreted as (map = (map = map)). You don't need to worry about this - as long as the headers of your special member functions are correct and you avoid self-assignment, this case will be automatically handled.
- Remember, you have iterators and the insert member function at your disposal!

Tips

Check out the Special Member Functions lecture and the Move Semantics lecture!

Try to use the **member initializer list** (colon after constructor) whenever possible for efficiency!

We do not provide the headers, so you will have to add them to the .h file yourself.

Read over the .h file again and make sure you understand each private member variable! You will need to fill these in your SMFs!

We have given you helpers which cover the vast majority of the actual nitty gritty of filling a copied/moved hashmap. Pay special attention to `insert()` and `clear()`!

Testing

You can run the Milestone 2 tests by uncommenting the lines marked "UNCOMMENT FOR MILESTONE 2" and then selecting option 2 when you run your code. Remember to `./build.sh` first!

You don't need to look at them, and the testing file actually contains more tests than we run, but Test 4A, 4B, and 4C create copies of your HashMap using the copy constructor and copy assignment operator, and verifies their correctness. In addition, there are tests for edge cases such as self-assignment. Tests 4D, 4E, and 4F try to move HashMaps from one to another using the move constructor and move assignment, and verifies correctness just like test 4A.

Note that the move operations and time tests may pass even if you haven't implemented the move operations (recall that if no copy/move operations are declared, the compiler generates default ones, which will pass the move but not copy tests). See the test_settings.cpp for more information. The tests use some compiler directives (#pragma) to silence compiler warnings about self-assignment. You can safely ignore those.

Test 4G and Test 4H time your move operations to verify that you are actually moving the contents of the HashMaps rather than copying the contents. The tests try the move operations on HashMaps of different sizes, and verify that the runtime of the move operations is O(1), not O(n). You can see the results of the time test printed in the test harness.

Writeup

Please answer these questions in `short_answer.txt`

1. Attachment Issues

   Why is it that we need to implement the special member functions in the HashMap class, but we can default all the special member functions in the HashMapIterator class?
   Hint: your answer should reference the Rule of Five (the Rule of 5 including move operations) vs. the Rule of Zero, and also talk about std::vector's copy constructor/assignment operator.

2. Move Semantics

   In your move constructor or move assignment operator, why did you have to std::move each member, even though the parameter (named rhs) is already an r-value reference?

Submitting

Please submit `hashmap.h`, `hashmap.cpp` and `short_answer.txt` to Paperless!