Assignment 2: HashMap

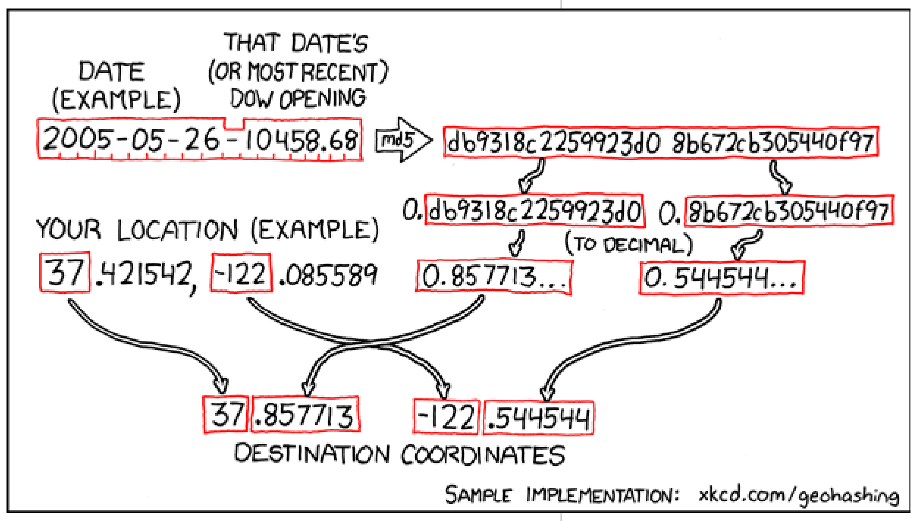
Checkpoint due: Thursday, March 11 at 11:59 PM

Rest of assignment due: Wednesday, March 17 at 11:59 PM

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. Implement a macro to hash string literals at compile-time.

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



Saturday is game night. *(Credit: https://xkcd.com/426/.)*

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 minimal implementation of a HashMap (the name of this data structure in the Stanford C++ library) that you could have written in CS 106B. The starter code is functional, 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.

You will not necessarily write a lot of code on this assignment (~50 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.

Tips for the Assignment (please read these)

Before we get started, here are a few things to keep in mind:

- Do not write any code that uses a pointer, directly changes a linked list, or interacts with the array of buckets (unless we explicitly ask you to). You should not need to call the private helper member functions either (find_node, make_iterator, etc.). Instead, use the iterators or public member functions (insert, find, erase, etc.). The assignment is much easier if you use the iterators, and we hope that you realize how powerful this the iterator/algorithm abstraction is.
- The starter code comments are thorough, more so than this handout. Read them!
- Want to test a function you wrote for the HashMap class? Use the student_main function, and call map.debug() to see the internal state of your class.

Assignment Timeline

The starter code passes 22 of the 36 required test cases. Your job is to get the remaining 14 test cases (6 in milestone 3, and 8 in milestone 4) to pass, and to reasonably answer the 10 required short answer questions throughout the assignment.

In the original version of the assignment, there were 10 milestones, and completing all milestones would help you accomplish #1. In the interest of time, and constrained by the fact that we only have 10 weeks, you only need to complete milestones 1, 3, and 4. We encourage you to go beyond the minimum requirements to create a truly STL-compliant template class, either now or in the future.

Milestones 2 and 5, as well as anything listed in the extensions, are completely optional. Milestones 2 and 5 form the remaining 7 optional test cases. You also do not need to answer the 1 short answer question in milestone 5.

Only Milestones 1, 3, and 4 are required, but we are happy to support you in completing 2 and 5. You don't have to do them in order, but finishing #1 will help you significantly with #3 and #4. **Please try your best to complete Milestones 1 and 3 in advance of the checkpoint deadline.**

1 6 questions, 0

Template classes, const-correctness, iterators, RAII.

2 10

Iterator-based constructors. (We provide you the .h file headers.)

3 2 questions, 15

Operator overloading. (We provide you all the headers.)

Checkpoint due Thursday, March 11 at 11:59 PM

4 2 questions, 35

Implement special member functions (headers not included.)

Final assignment due Wednesday, March 17 at 11:59 PM

5 1 question, 0

Implement benchmark tests.

Starter Code

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

- `hashmap.h`: This file contains the class function prototypes for the minimalistic HashMap, and also contains thorough comments about each function. **The prototypes for milestone 3 are provided here, but you will need to add your function prototypes for any function that you write for milestone 4.** Like in CS 106B, you may add any private helper functions, 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. The function skeletons for milestone 3 are provided here which mirror the prototype in the .h file. **You will need to put the implementations for the functions for milestones 3 and 4 in 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.
- `student_main.cpp`: When the test harness is turned off (see below), the student_main() function in this file will be run. You can treat this as if you were writing your own main function to test your program. You are encouraged to write some code here to test and play around with the HashMap implementation, but it is not necessary to modify this file.
- `test_settings.cpp`: This file contains macros which lets you on the test harness. If you turn on the test harness (#define RUN_TEST_HARNESS 1), test cases will be run. If it is turned off (#define RUN_TEST_HARNESS 0), the student_main function will be run. This is analogous to using NO_TESTS vs. ALL_TESTS in CS 106B's SimpleTest. The file also lets you turn on or off individual test cases. Test cases that are turned on will result in a PASS or FAIL when run. Test cases turned off always result in a SKIP (which is effectively a FAIL). The reason you will turn off test cases is that the test cases for milestones 2 and 4 may call functions that you have not declared or implemented, so turning them on will generate compiler errors. Only turn on a test case after you have implemented that function, as explained in the test_settings.cpp file. You may get compiler errors as well if your function prototype is incorrect (likely because of const correctness, or parameter/return value type issues).
- `tests.cpp`: This file contains the code for each test case. You are welcome to look at this, add some map.debug() statements, or even add your own test cases. We will test your code on an unaltered version of this file, so be careful about changing this file.
- `short_answer.txt`: You will write your solutions to the short answer questions in this file.
- `HashMap-V4.pro`: This file contains the Qt settings for the project. You will notice a flag -O0 in this file, which controls the amount of compiler optimization. In milestone 5 you will change this flag to -O2 when running benchmark tests. You do not need to modify this file, unless you decide to work on optional milestone 5.

Background Reading and Examine Starter Code

You will implement the equivalent of the Stanford HashMap, using a hash table that resolves collisions using external separate chaining. Luckily for us, CS 106B covers hashing and external hash tables with chaining! Before reading further, familiarize yourself with hashing by reading the resources below (if you're in CS 106B or recently took it, feel free to skip the CS 106B resources). Note that for the purpose of this assignment, we have implemented each bucket in the HashMap as a **linked list** rather than using another data structure as you may have seen in CS 106B.

The Map Interface

- Stanford documentation of HashMap.
- STL documentation of std::map and std::unordered_map (based on a hash-table).
- CS 106B (Fall 2020) notes on Maps.

Hashing

Note: some authors insert a node to the end of a bucket's linked list. We will always insert into the front, since order doesn't matter and inserting to the front is faster and easier.

- A YouTube Video
- CS 106B (Winter 2021) notes on hashing here and here.
- Avery Wang's (2019-2020 106L lecturer) recorded session on hashing (optional if you're feeling a little shaky on hashing after reading above resources)

Milestone 1: Short-Answer Questions + Reading the Starter Code

Read through the starter code's interface (.h) and template implementation (.cpp) files to understand what has already been implemented. This handout you are reading is fairly sparse, and the reason is that there is a lot of documentation provided in the starter code. You should be able to identify the standard map interface (constructor, insert(), at(), contains(), etc.) as well as the hash table implementation (_bucket_arrays, node*'s, etc.).

Go to the student_main function, and try solving your favorite CS 106B HashMap problem there. You can call map.debug() inside the test cases to see a visual representation of the map printed to the console. You can call this function in your student_main function without the test harness, or inside the test cases.

Then, navigate to `short_answer.txt` and answer these short-answer questions:

1. Templates and Template Classes

In the `rehash()` function, the for-each loop contains an auto&. What is the deduced type of that auto, and why the ampersand necessary?

2. HashMap Pair Type

STL containers store elements of type value_type, and for your HashMap this value_type is a std::pair<const K, M>. What would be the problem in the HashMap class if value_type were instead std::pair<K, M>?

Hint: think about the following lines of code:

```
HashMap<std::string, int> map;
map.insert({"Avery", 3});
auto iter = map.begin();
iter->first = "Anna";
auto anna_iter = map.find("Anna");
```

3. 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.

4. RAII?

This HashMap class is RAII-compliant. Explain why. (If you haven't seen this yet, wait until the appropriate lecture comes up in class.)

HashMap Iterators

In a previous version of this assignment, we asked students to implement an iterator class. By implementing an iterator class, you have a simple way of traversing the elements in the HashMap. Combined with iterator-supported implementations of find, insert, and erase, you never have to manipulate the linked lists in the later milestones. However, implementing an iterator class is fairly time-consuming, so we decided to provide you our implementation of the iterator class and the HashMap member functions that interface with the iterator class (e.g. begin(), end(), find(), etc.). Instead, you should read and understand how the iterator class is implemented, and answer short answer questions on it.

5. p r i v a c y

Why is the HashMapIterator's constructor private? How do HashMapIterators get constructed if the constructor is private?

Hint: for the second question, HashMap and HashMapIterator class are mutual friends of each other. You'll still need to explain why that matters - check out where the constructor of HashMapIterator gets called inside HashMap.

6. Increments

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

(Optional) Milestone 2: Iterator-Based Constructors

This milestone is **optional**. This milestone naturally fits here after you have explored the iterator class, and are ready to practice using the iterator class while also learning some important new C++11 class design syntax.

The starter code already provides multiple constructors. We will add two more iterator-based constructors:

- Range constructor: given a range (i.e. [start, end), where start and end are iterators) of elements, construct a HashMap whose elements are precisely the elements in this range.
- Initializer list constructor: given an std::initializer_list of elements, construct a HashMap whose elements are precisely the elements in this std::initializer_list. This is the constructor called when you use uniform initialization to directly specify the initial elements.

The headers for each constructor is provided to you in the .h file. You will need to write the function itself in the .cpp file. The implementation is not difficult (~10 lines total), but we're also introducing a few new language features and syntax so you may need to read the C++ documentation.

Here are some hints:

- You can have one constructor call another as a **delegating constructor**. These two constructors are very similar, and in fact you can have the initializer list constructor call the range constructor. You can even have the range constructor call the other constructors in the starter code.
- You should take a look at **std::initializer_list**. Notice that you can only access the elements by operating over the iterators.
- Use the insert member function. Do not try to modify the linked list yourself.
- The range constructor is a template function inside of a template class, since you have an extra template parameter InputIt that is not a template parameter of the class. The syntax is tricky - check it out here.

Milestone 3: Operator Overloading

You will implement four operators. The headers are provided in the .h file, and the function skeletons are provided in the .cpp file. Make sure you undestand why these skeletons and headers are consistent with the principles taught in the operator overloading and const-correctness lecture.

Indexing (`[]`)

The index operator accepts a key, and returns a reference to its mapped value. The index operator should support **auto-insertion**—if a key is not found, then a new key/mapped pair is inserted with the given key and a default value of the mapped type.

Hints:

- This function can be implemented in a single line, making a single pass through a linked list. **Specifically, think about what the insert member function already does**: what does it do if the key already exists, and what does it do otherwise? Does the behavior match with what you need?
- The default value of the mapped type is simply the type you get by calling that type's default constructor (i.e. `M val;`).
- You can use uniform initialization from C++11 to call the default constructors of the types that are automatically inferred. For example, if a parameter to a function foo is std::pair, you could write foo({ {}, {} }), which would call the pair constructor with two arguments, {} and {}. The first {} would call the default constructor for int (which would just be 0), and the second {} would call the default constructor for std::string (which would be ""). You don't need to specify the types, the compiler will automatically infer it for you! Isn't uniform initialization awesome?
- You need to return a reference to the key/mapped pair that is inside a node in the hash table stored in heap memory. Be careful to not return a reference to a local variable that is residing on the stack. See the starter code comments for an example of what NOT to do.

Stream Insertion (`<<`)

The stream insertion operator writes the contents of a HashMap into an output stream. The format resembles the format used by the Stanford HashMap, as seen below:

```
HashMap<string, int> map;
cout << map << endl;        // prints "{}"
map.insert({"Anna", 2});
cout << map << endl;        // prints "{Anna:2}"
map.insert({"Avery", 3});
cout << map << endl;        // prints either "{Avery:3, Anna:2}"
                            // or "{Anna:2, Avery:3}"
```

There should be a comma and a space between each pair, but not before the first pair or after the last pair.

The pairs can be in any order, since this is, after all, an unordered map.

The stream insertion operator will need to support chaining.

Hints:

- We suggest building up your string using a **stringstream**. This allows you to easily insert an element into the stream, which will convert it to a string for you. This will be especially helpful if you have extra characters in your representation at the end. We didn't talk about stringstreams extensively in class, but they are very similar to all of the streams we have discussed. See here for a code example. Also, note that you can use the str() method on a stringstream to convert its contents to a string.

- Chaining will work correctly as long as your return value is correct.
- How do you iterate through all the elements? Since iterators are implemented, a for-each loop will work as you expect! See the lecture on iterators if you forgot the connection between iterators and for-each loops.

Equality (`==`) and Inequality (`!=`)
Two HashMaps are considered equal if they have the exact same key/mapped pairs. The internal order that they are stored, or the number of buckets, does not matter.

Hint: You should only have to loop through one of the HashMaps once.

Test Cases

Test 2A, 2C, and 2E test the functionality of the indexing, stream insertion, and equality/inequality operators, respectively. Tests 2B, 2D, and 2F test the const-correctness of the respective operators. Since we've given you the correct headers, unless you changed the header, tests 2B, 2D, or 2F should pass if 2A, 2C, or 2E pass, respectively.

Fun fact about the test harness: the const-correctness tests check whether proper use of the HashMap class as a client compiles. It also uses some advanced C++ metaprogramming techniques to check that improper use of the HashMap class as a client does not compile. In past quarters, students had to come up with the headers of these operators themselves, so this would check whether they have const-correctness issues.

7. Did We Make a Mistake? (Hint: no)

Why is there both a const and non-const version of at(), but only a non-const version of operator[]? (unlike in Vector where operator[] also had a const version).
Hint: `operator[]` has some extra functionality compared to the `at` function.

8. Now Streaming on iTunes

Look at the function signature for the stream insertion (operator<<) in hashmap.cpp. Briefly explain the syntax for this function signature and how this works.
Hint: your answer should reference (no pun intended): the parameters and return value, their constness, and the fact that they are references; and the reason why this operator is not a member function, and not a friend of HashMap (it's not even declared in the .h file!).

Milestone 4: Implement Special Member Functions

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.

Tips

We are requiring you 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.

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 nullptrs. 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!
- 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.
- Just like in milestone 3, you have iterators and the insert member function at your disposal.

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.

Now, answer these questions:

9. 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 3 including move operations) vs. the Rule of Zero, and also talk about std::vector's copy constructor/assignment operator.

10. 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?
Hint: the lecture on move semantics goes into this in some detail.

(Optional) Milestone 5: Benchmark Performance

This section is *optional* but shouldn't take that long and should be interesting, as well as help you understand efficiency.

We've written some performance tests for you. Follow the instructions below.

- Turn on the benchmark tests (in the test_settings.cpp), and run the program. Note that the runtimes look odd for small inputs.
- Go into the benchmark tests named "benchmark_insert_erase", "benchmark_test_find", and "benchmark_test"iterate" and change the number of buckets from 500000 to 'size'. Rerun the program, and you will see the results of all three benchmark tests.
- Go to the HashMap-V4.pro file and change the -O0 to -O2, and run the program in "Release" build. The compiler will now fully optimize the code.

Some optional questions for you to think about (do not submit your answers to these):

- What's the difference changing the number of buckets from 500000 to 'size'? Why?
- Looking at the benchmark test results, what is the Big-O complexity of insert and erase? Feel free to ignore the rows where the input size is 100k and 1M.
- Do you notice any noticable differences between the insert/erase benchmark, the find benchmark, and the iterate benchmark, in comparison to std::unordered_map? Why do you think that is the case?
- What happened when you turned on the compiler optimization?
- (beyond the scope of 106B, this is a computer architecture question) Why does the runtime explode when the input size hits 1M?

Our HashMap is faster than `std::unordered_map`! That's certainly unexpected. There are many more HashMap implementations which can perform faster than our HashMap. Check out this link.

Submitting the Checkpoint

For the checkpoint, you should submit Milestones 1 and 3 in **two files:** `hashmap.cpp` and `short_answer.txt`. If you worked with a partner, please have one partner submit the assignment. There will be a place on Paperless for you to enter in the other partner's SUNet ID so that you both get credit.

Submitting the Final Project

When you're finished, submit **three files:** `hashmap.cpp`, `hashmap.h`, and `short_answer.txt` on Paperless! Follow the same instructions for partner work as above.

And finally... Good luck! :)

Extensions: Creating a Fully STL-Compliant HashMap

All of these are optional. Some may require supplemental material or additional reading.

Implement try_emplace

The insert member function creates many unnecessary copies. At the function call, it first creates a temporary copy of the key and the mapped value, then it creates a temporary pair containing that key and the mapped value. Inside the insert function, it creates a copy of this pair (since it's passed by const l-value reference), and then it assigns the value_type in a node to be a copy of this pair.

You've solved a similar problem when learning about move semantics. There is a more advanced move semantics problem called perfect forwarding, which uses a combination of variadic templates, r-values, and a new function std::forward. Read about emplacement here, and then implement the member function try_emplace that the std::unordered_map has.

Implement a LinkedHashMap

If you were disappointed that you did not get to implement your own iterator class, you should try creating a more advanced iterator class! Instead of iterating through the elements in the order they are stored in the HashMap (i.e. unordered), you should iterate in the order of insertion. This is a tricky linked list problem, as there are two linked lists - one for the buckets, and one to keep track of order insertion. Java has a LinkedHashMap data structure that C++ does not have. Implement one!

Automatic Rehashing

You should have realized in the benchmark tests that one main drawback of our current HashMap is that it does not automatically resize. Implement that feature in your HashMap! Experiment with what kind of rehash policy works best on a variety of workloads.

Make the HashMap more perfect

Implement any features our HashMap is missing. Check out the std::unordered_map page for more ideas!

Implement (ordered) map

Implement the same interface, but backed by a binary search tree instead! To make iterator traversal easier, add a "parent" pointer to each node in addition to the "left" and "right" children.

Implement your own containers

In the past we've had students implement Kd-trees, Gap Buffers, and Merkle Trees. Find one you really like, implement it, and add some iterator support!