CS106L Spring 2020 REVISED EFFECTIVE MAY 31

Assignment 2: STL HashMap

Assignment created by Avery Wang (lecturer for 2019-20)
created with input from Anna Zeng.

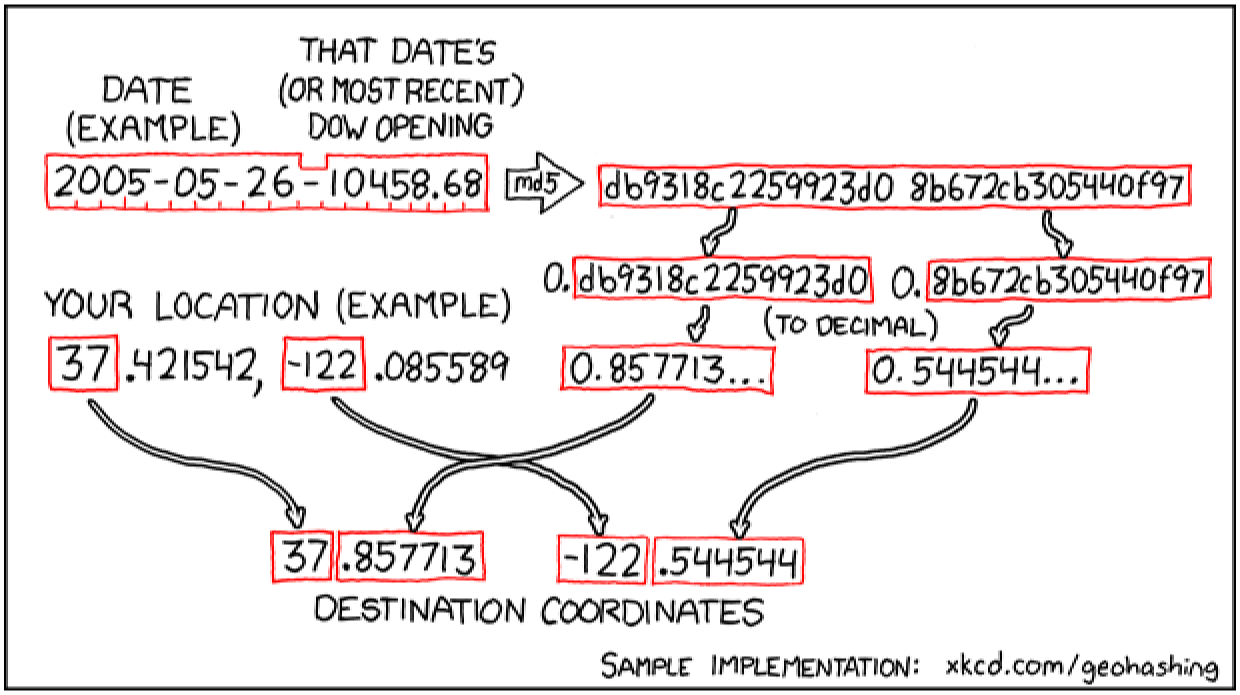
According to veteran C++ programmers, there are two projects which bring together all the knowledge that a proficient C++ programmer should have.

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

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, 2, 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.

You will not necessarily write a lot of code on this assignment (~100 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.

See page 3 for deadlines.



Saturday is game night.

Introduction

In the second week of CS 106B, you were introduced to the Map abstract data type. A map stores key/value pairs, and provides efficient lookup of the value of any key.

The `std::map` was traditionally implemented using a balanced binary search tree, which you'll learn about in CS 106B week 8. C++11 brought us the `std::unordered_map`, which uses a clever technique called hashing and supports the most map operations in constant time (!). In this assignment, you are 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, yet 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 might be wondering, why are we reimplementing something that is standardized in the STL? Implementing an STL-compliant `HashMap` container is a very good exercise because…

- this forces you to understand how the STL was designed, so that you will be able to use the STL most effectively as the client.
- you will use literally everything you learned from the class, from structs and references, functions and algorithms, everything in template class design, all the way up to the latest lecture in move semantics.
- while the STL containers are great for general-purpose programming, people (in "the real world") absolutely implement their own containers. Let's say you were a data engineer. You may want to implement a HashMap that takes advantage of something specific to your domain (like cache locality). Say you were a Aero/Astro PhD trying to develop safety-critical code for your aircraft avoidance system. The STL containers aren't safety-critical, so you will have to develop your own using the same techniques. Are you writing optimized code for machine learning? Most machine learning frameworks use specialized data structures that allow vectorized operations, which surely do not use the standard provided data structures. The parts of Pytorch written in C++/CUDA probably didn't use the STL vector. They wrote their own specialized vector. Fun fact: the three students here were past CS106L students trying to apply C++ to their own domain.
- we chose a familiar data structure so that you can focus on C++-specific design and not have to learn a brand-new data structure. If you want a challenge, we can assign you alternative data structures (Gap Buffers and Kd-trees) that you will have to research, learn about, and code before making them STL-compliant.
- for those of you in CS 106B, this is great linked list practice.

Assignment Timeline

To help you stay on track for this assignment, we will require you to submit checkpoints each week. Late day policies apply wherever we say "deadline enforced", although as always we will grant extensions up to the hard deadline.

Checkpoint: pass all starter code + milestone 1 tests.
Due Date: Tuesday, June 2, at 11:59 pm (deadline enforced)
Deliverables: hashmap.h, hashmap.cpp.
If you already submitted checkpoint 1 previously, you don't have to submit again.

Final submission: pass milestone 1-2 tests + submit short answer questions.
Due Date: Sunday, June 7, at 11:59 pm (deadline enforced)
Hard Deadline: Wednesday, June 10, at 11:59 pm (deadline enforced)
The hard deadline is strict by university policy. No extensions will be granted.
Deliverables: hashmap.h, hashmap.cpp, short_answers.txt.
Submit any extensions, such as the iterator file, your main program, test cases, etc.

Summary of Milestones

The milestones in green are required, and the milestones in red are optional. All milestones must be done in order, as later milestones will depend on earlier ones.

Milestone	Lines of Code	Description
0	N/A	Read about hashing. Examine starter code/test harness.
1	~10-15 lines	Complete the implementation of rehash.
2	~40-50 lines	Overload 4 operators and make Hashmap const-correct.
3*	~30-40 lines	Implement 4 special member functions.
4	N/A	Answer 10 short answer questions about your code.
5	~15 lines	Implement initializer_list and range constructors.
6	~120 lines	Implement an iterator class.
Beyond?	???	See page of extensions.

*made optional effective May 31

Grading and Feedback

To pass this assignment, you must pass the 13 starter, milestone 1, and milestone 2 test cases, and reasonably answer all 10 short answer questions. We will give a code review to what you submit by the deadline.

Tips for the Assignment

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

- Milestone 0 requires knowledge of linked lists, which CS 106B covers in week 7. You should have plenty of time to complete the assignment due in week 10.
- The starter code comments are thorough, more so than this handout. Read them!
- The handout may seem vague as in describing what you are supposed to do. We don't provide even the function prototypes. **This is intentional**. Designing the class definition itself is a valuable learning activity, and we don't want to constrain you to one particular design. You need to consider how a client might use your HashMap, and the test harness will try every reasonable way of using your class. Ask us if you are ever unsure about how to proceed.
- Avery has made three different videos on hashing, milestone 2 interface design, and milestone 4 design short answers. If you are stuck, watch these videos linked in the relevant pages below.

Downloading the Starter Code

Option 1 (Qt Creator): Visit the CS 106L class website, go to the assignments tab, and download the zip file that contains the Qt Creator project.

Option 2 (Download zip): Visit the CS 106L Github repo, click the green button that says "Clone or Download", and click "Download ZIP".

Option 3 (Fetch directly from command line): If you haven't cloned our CS 106L repository before, type the following into the command line, and you'll get the assignment starter code, as well as all previous lecture code.

```
git clone https://github.com/averyw09521/CS106L-spr20.git
```

If you've cloned the repository before, simply type "git pull" to get the assignment files.

Starter Code Provided Files

hashmap.h

This file contains the function prototypes for the minimal `HashMap`, and also contains very thorough comments about each function. The prototypes for `rehash()` are provided here, but you will need to add your function prototypes for any function that you write for milestone 2 (and optionally 3).

hashmap.cpp

This file contains the implementation of the HashMap, and also contains very thorough comments about implementation details. The skeleton for `rehash()` is provided here, and you will need to declare your own functions for milestones 2 (and optionally 3).

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.

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.

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 3 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.

How do you run your program? In Qt Creator, just press the "Run" button. On the command line, type in "`make`", followed by "`./main`".

Milestone 0: 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. Before reading further, familiarize yourself with hashing by reading the resources below.

The Map Interface

- Stanford documentation of HashMap.
- STL documentation of std::map and std::unordered_map (based on a hash-table).
- This quarter's CS 106B 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
- MIT OCW: very detailed explanation, although you don't need to understand any of the analysis of the runtime at the end.
- CS 106B (Fall 2018 slides)
- CS 106B (Winter 2020 slides): remember that in a map, we are storing pairs of keys and values.
- **Link to the recorded help session.**

Reading the Starter Code Documentation

We could have asked you to implement a `HashMap` yourself given you knowledge from CS 106B. However, since practicing linked list algorithms is not the goal of CS106L, we'll give you 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.). You can call `map.debug()` inside the test cases to see a visual representation of the map printed to the console. This will be useful when you begin milestone 1.

Milestone 1: Implement `rehash()`

You will first implement one public member function. Its expected behavior is documented in the .h file. Your implementation of `rehash()` must not allocate or leak memory. Instead, `rehash()` should reuse existing nodes.

The purpose of this milestone is to get you familiarized with the starter code and with working in a template class. It should also serve as a good review of the linked lists algorithms you learned in CS 106B. Our solution is 11 lines of code long. If you are unsure about the hash functions, we recommend looking at the implementations of `insert()` and `erase()`.

Once you are done, turn on tests 1A and 1B. Test 1A checks the external correctness of the rehash function, by verifying that your HashMap is still functional even if we rehash to various bucket sizes, including extremely large or small values. We also interweave the calls to `rehash()` with calls to `erase()` and `insert()`. Finally, there is a check to see if you correctly throw an exception when `rehash(0)` is called (this was done for you in the very first line of rehash).

Test 1B roughly checks the internal correctness of the `rehash()` function. The idea is that buckets with longer linked lists should take longer to search, so we time various calls to `contains()` to verify that the sizes of the linked lists are roughly correct. This test is fairly rough as it depends on the speed of your computer, but we give it a large enough leeway that you should be able to pass the test. *It is fine if this test fails with a small probability.*

Milestone 2: Operator Overloading and Const-Correctness

You will implement four operators, and then you will make your `HashMap` `const`-correct. Here is the link to the recorded help session.

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.

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`.

- Print all key mapped pairs in the format `"key:mapped"`.
- The pairs can be in any order, since this is, after all, an unordered map.
- The list should begin with an open curly brace, and end with a closed curly brace.
- There should be a comma and a space between each pair, but not before the first pair or after the last pair.

```
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}"
```

The stream insertion operator will need to support chaining.

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.

Const Correctness

Finally, make your `HashMap` class `const`-correct. You will need to decide which functions (including those part of the starter code) and operators need to be modified.

After implementing an operator, turn on that operator's tests. There is one test for each category of operator, and one for const-correctness. If you find compiler errors appear after enabling Test 2D, this means that either one of your prototypes is incorrect (eg. wrong const-ness), or your `HashMap` class is not `const`-correct and needs to be fixed.

Milestone 3: Implement Special Member Functions (Optional)

The following functions were implemented for you in the starter code:

- Default constructor
- Destructor

Your job is to implement the following functions:

- Copy constructor
- Copy assignment operator
- Move constructor
- Move assignment operator

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`. Avoid memory leaks and perform your copy/move as efficiently and safely as possible.

Test 3A creates 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. Test 3B tries to move HashMaps from one to another using the move constructor and move assignment, and verifies correctness just like test 3A. Note that Test 3B passes even if you only implemented the copy operations (recall that if no move operations are declared, the compiler defaults to the copy operations). Test 3C times your move operations to verify that you are actually moving the contents of the HashMaps rather than copying the contents.

The tests use some compiler directives (`#pragma`) to silence compiler warnings about self-assignment. You can safely ignore those.

Milestone 4: Answer Short Answer Questions

Answer the following questions in the file named "short_answers.txt". These questions do not require long answers or even complete sentences. We will grade for reasonable correctness and effort. We went over the answers to these in a recorded session available at this link.

Example

Question: Consider the test case `B_move_ctor_assignment`. You should see a line with `std::move(map1)`. Then, in the next line, we move from `map1` to create `move_constructed`. Why is this valid?

Answer: `std::move` marks map eligible to be moved, but does not actually move. A function taking an r-value (eg. move constructor) does the moving. `map1` is not passed to such a function, so we can still move from `map1`.

Templates and Template Classes

1. In the `clear()` function, the for-each loop contains an `auto&`. Explain what the deduced type of that `auto` is and why the ampersand is necessary.

2. Take a look at the code inside the test `F_custom_hash_function()` of the file `tests.cpp`. What is the purpose of `decltype`, and is there a way to write that line of code without using `decltype`? Explain.

3. 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>`?

4. Take a look at the function `check_map_equal()` inside `tests.cpp`. You might notice that in the test cases, we use `check_map_equal()` to compare a `HashMap` and a `std::map`, but never to compare two `HashMaps` (we instead use `operator==`). Why can't we use `check_map_equal()` to compare two `HashMaps`?

Operator Overloading and Const Correctness

5. Why did you implement 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)

6. Out of all the operators you overloaded, `operator<<` is the most interesting.
    - Did you implement `operator<<` as a member, a friend, or neither? Why?
    - We had you overload `operator<<` as an exercise in operator overloading, but none of the STL collections have `operator<<` overloaded. Why do you think the STL designers made this decision?

7. If you had to implement an iterator class for your `HashMap` (actually implementing one is optional, see milestone 6), what would its iterator category be and why?

Special Member Functions and Move Semantics

Even though milestone 3 was made optional, you should still answer these as if you hypothetically implemented the special member functions. See video on milestone 4 design questions if you aren't sure about these since you didn't implement them.

8. Consider a class `MyClass` which has three private members - a `size_t`, a function object, and a `std::vector<int>`. In this case the rule of zero applies and you should not write your own special member functions. In your `HashMap` class the rule of five applies - the starter code had the destructor, and you implemented the other four special member functions.
    - Why does the rule of zero apply to `MyClass`?
    - Why does your `HashMap` class follow the rule of five and not the rule of zero, if your `HashMap` private members are also a `size_t`, a function object, and a `std::vector`?

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

RAII

10. Is your `HashMap` class RAII-compliant? If so, explain. If not, give a specific instance in which this could be a problem, and explain how you would make your class RAII-compliant.

Extensions: Creating a Fully STL-Compliant HashMap

All of these are optional. Some may require supplemental material or additional readings. We encourage you to attempt these, even after the end of the quarter.

We provide an autograder for milestones 5 and 6. These tests are slightly less polished, but if you find any bugs, please let us know!

Milestone 5: Implement an initializer_list constructor and a range constructor.
Add a constructor that allows you to construct your `HashMap` with an `std::initializer_list`. Also add a constructor that accepts a range (in the form of two iterators) to initialize your `HashMap`. This is a fairly easy extension that can be implemented in around 15 lines of code.

Milestone 6: Design an Iterator Class for your HashMap
Using your response to question 7 in Milestone 4, implement an iterator class for your `HashMap`. Then, add support for `const_iterator`, and make sure that you can create `const` iterators, `const_iterator`, and `const const_iterator`. To reduce the repeated code, you may need to do some additional reading about `type_traits` and template metaprogramming. Talk to us if you want some guidance on this. Once you are done, change HashMap.cpp to use iterators. The code is much simpler with iterators!

More ideas (originally these were part of the assignment):

- Implement iterator based versions of `find()` and `erase()` (milestone 7)
- Implement r-value overloads of various member functions (milestone 8)
- Incorporate smart pointers into the linked lists (milestone 9)
- Implement `try_emplace()`, which will require knowledge of *variadic templates* and *perfect forwarding*. (originally milestone 10)
- Equip the HashMap with a custom memory allocator.
- Add template support for equivalences relations.
- Make some functions constexpr, add some compile-time optimizations.
- Implement a LinkedHashMap, where iterators traverse the map in the order the elements were inserted.
- Support automatic rehashing when the load factor is too high.

And beyond?
Every so often, we think of something new that makes our class more complete. The most recent addition is a converting constructor from `iterator` to `const_iterator`. If you come up with anything interesting, talk to us!

Alternate Final Project

You can choose to complete your own final project. You should design and implement a class that...

- is a template class.
- is non-trivially const-correct.
- supports at least two categories of operators (eg. comparison and indexing)
- supports non-trivial special member functions (i.e. rule of five needs to apply) and move semantics. The resource you are handling does not have to be memory, though that is likely the simplest option.

The easiest way to satisfy all these requirements is to implement a more advanced data structure, such as a Gap Buffer, a Kd-Tree, or anything else. The heap-based priority queue from CS 106B is not sufficient, although more advanced heaps (eg. binomial heaps) may be acceptable.

Be aware that this will almost certainly be much more work than the default HashMap project. We worked really hard to streamline the default project and gave a ton of starter code so students don't have to write code not relevant to CS 106L. Since a custom project will have no starter code, you'll have to implement the logic of the data structure (like CS 106B's priority queue assignment) before incorporating the ideas of CS 106L.

If this is a path you are interested in, follow the deadlines below.

We have reached out to students who are completing a final project to determine an appropriate alternate schedule.

When grading, we will focus more on design and effort, rather than a strict test of correctness. While correctness is still important, it's difficult to test since there is no testing harness. We just want to see that you've learned and practiced the CS 106L material.

You're done! We enjoyed teaching you this quarter.
Thanks for the fun quarter, and have a great summer!

Update Log (major changes will be emailed):
- 5/31: In response to the difficult circumstances the world and in particular the U.S. has been going through, we have made milestone 3 optional, and created more support for students who are stuck.
- 5/22: We just wrote the grading rubric, so I added some info on how we'll give feedback for this assignment, with the bucket grades.
- 5/21: Fixed the deadlines in the alternate project to match the ones in the default project (which are later than the original one we listed for the alternate project).
- 5/21: Rephrased what an enforced deadline for checkpoint 2 is, and the late day policy, but also reiterated that we will grant extensions up to the hard deadline.