CS 251 : Data Structures
Spring 2020

Project #06: Part 1 of 2: hashmap

Complete By: **On-time:** Friday 4/10 @ 11:59pm
Late: Saturday 4/11 @ 11:59pm (-10%)

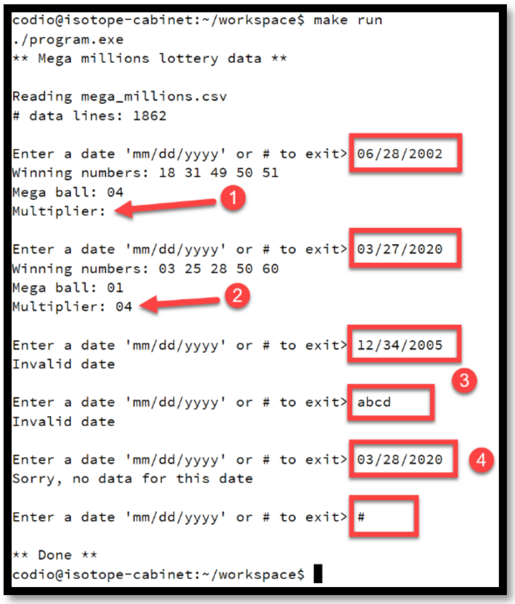
Assignment: "hashmap.h" file

Policy: Individual work only, late work *is* accepted

Submission: "hashmap.h" file via Gradescope; the first 12 submissions are free, each additional submission 1 pt

Overview

The goal is an extensible **hashmap<KeyT, ValueT>** class that implements hashing, handling collisions using probing. The hashmap represents part 1 of the project. In part 2 of the project, you'll use your hashmap to analyze data from the **Divvy** company. This handout focuses on part 1 only.



```
codio@isotope-cabinet:~/workspace$ make run
./program.exe
** Mega millions lottery data **

Reading mega_millions.csv
# data lines: 1862

Enter a date 'mm/dd/yyyy' or # to exit> 06/28/2002
Winning numbers: 18 31 49 50 51
Mega ball: 04
Multiplier:

Enter a date 'mm/dd/yyyy' or # to exit> 03/27/2020
Winning numbers: 03 25 28 50 60
Mega ball: 01
Multiplier: 04

Enter a date 'mm/dd/yyyy' or # to exit> 12/34/2005
Invalid date

Enter a date 'mm/dd/yyyy' or # to exit> abcd
Invalid date

Enter a date 'mm/dd/yyyy' or # to exit> 03/28/2020
Sorry, no data for this date

Enter a date 'mm/dd/yyyy' or # to exit> #

** Done **
codio@isotope-cabinet:~/workspace$
```

On the course dropbox (and Codio), a solution is provided to the lab from week #11. Recall that this was the lottery data program. The provided solution is written using a **hashmap** class, much like the one you need to write. In fact, your solution here in part 1 should start with the provided implementation. The hashmap class allocates the hash table, and provides the **insert()** and **search()** functions. Interestingly, the **Hash()** function remains separate and *outside* the class, allowing the hashmap to be used in other scenarios. Here's code for main() rewritten to use hashmap:

```
hashmap<string, LotteryData> hmap(37200);

.
. // insert data into hmap from csv file:
.

cin >> thedate;
while (thedate != "#")
{
   LotteryData  ld;

   if (hmap.search(thedate, ld, Hash))  // notice Hash() function passed as param:
   {
      cout << "Winning numbers: " << ld.Numbers << endl;
      cout << "Mega ball: "       << ld.MegaBall << endl;
      cout << "Multiplier: "      << ld.Multiplier << endl;
   }
   else // not found:
   {
      cout << “Sorry, no data for this date” << endl;
   }

   cin >> thedate;
}//while
```

Review the provided code for the complete details, but the keys aspects to notice are the following:

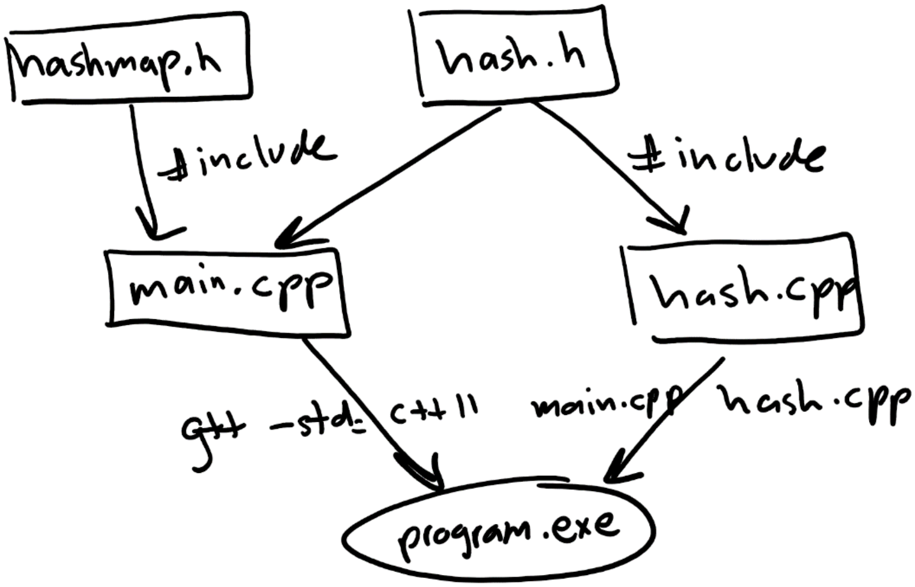
1. The hashmap stores the (key, value) pairs
2. The hashmap does hashing inside the insert() and search() functions
3. The Hash() function is passed as a parameter to the insert() and search() functions

The last one is the most interesting, since this means you can have different hashmaps storing different data using different hash functions. That’s what we meant by an “extensible” hashmap class.

Getting started

You’ll want to review the provided solution carefully. As we start to program using multiple .h and .cpp files (which is how most C/C++ programs are written), you’ll want to appreciate --- and correctly utilize --- this approach. Build the program and convince yourself it works. You can even submit the program to Gradescope (“**Lab Week 11 – testing only**”) and it should score 100.

The provided files are available on the course dropbox under Projects, project06-files. A Codio project is also setup: **cs251-project06-hashing**. The provided solution consists of two .h files and two .cpp files:



```
g++ -std=c++11 main.cpp hash.cpp
```

As you work with multiple files, keep in mind that we are building *one* program. So any function in any C++ file can be called from another other C++ file. Example: the "hash.cpp" file contains 2 functions: **isNumeric()** and **Hash()**. Folks wanted to take advantage of the isNumeric() function in "main.cpp" --- which is great. But the mistake is that they copy-pasted the code from hash.cpp to main.cpp. Whenever you copy-paste, an alarm should go off in your head --- "this is the wrong approach". In this case, since the function already exists in "hash.cpp", what you want to do instead is just call it from "main.cpp". To do that, you make it available by placing the function header in "hash.h":

```
int isNumeric(string s);
```

Now you just call the function from main.cpp. [*BTW, this is why .h files are called header files, they typically contain function headers.*]

You might be wondering, "**Why does hashmap.h contain both function headers *and* C++ code? Why isn't the C++ code for hashmap in a separate .cpp file?**" Great question. Since hashmap is templated, the C++ code must reside in the .h file; it's a current limitation of C++. Normally, classes follow the same rules as other files: header information in the .h file, and C++ implementation code in the .cpp file. But templated classes are an exception.

Assignment --- handling collisions

Your assignment here in part 1 is to re-write the **hashmap** class to support collisions using probing. In other words, from the outside --- when called from main() --- the hashmap class should behave exactly as before. The **insert()** function inserts, and the **search()** function searches. The fact that collisions are occurring should make no difference to the main() program.

However, the provided hashmap class is not written to handle collisions; it assumes a perfect hash function. If the Hash() function causes collisions, the hashmap will fail to function properly (overwriting values, returning wrong values, etc.). This implies you need to rewrite the **insert()** and **search()** functions to handle collisions using probing.

Recall that lecture 29 (Friday 4/3) discussed how to handle collisions using chaining and probing. You should watch or re-watch this recording before starting to program. The recording can be found on BB under "lecture recordings".

This is not meant to be a difficult exercise; perhaps 50 lines of code. In fact, the only changes required are to the following aspects of "hashmap.h":

1. *Update the header comment since it's misleading. The class does not require a perfect hash function, and the class now handles collisions. Add your name while you're there.*
2. *Update the insert() function to properly handle collisions using probing. Note that if the (key, value) is already in the hash table, the existing value is overwritten by the new value. You may assume the hash table will never fill, and you will eventually find an empty location to insert.*
3. *Update the search() function to properly handle collisions. You may assume the hash table will never fill, and you will eventually reach an empty location.*
4. *Add a copy constructor to properly make a deep copy.*
5. *Overload operator= to properly free memory and make a deep copy.*

How to test? Interestingly, all you need to do is start generating collisions, and see how the program behaves. If the program behaves exactly as before then the hashmap is handling collisions properly. If not, then something is wrong. The easiest way to generate collisions is to reduce the size of hash table, in this case to a value < 37,200. The smaller the size, the more collisions (although be sure there's enough room to insert all the data). There are roughly 2,000 lines of data in the .csv file, so start by trying maybe a size of 30,000. A simple change to the top of main() is all you need:

```
int main()
{
   cout << "** Mega millions lottery data **" << endl;
   cout << endl;

   //
   // Use a smaller array size now to generate collisions:
   //
   const int N = 30000;
   hashmap<string, LotteryData> hmap(N);
```

Run and test locally, entering a few dates. The program should work as before. Then submit to gradescope ("**Lab Week 11 – testing only**"), and the score should remain a 100. Repeat the exercise by reducing the array size further. As long as N > 2000, the program should work as before --- the only difference is that it's running slower as the # of collisions increase.

As for testing the copy constructor and operator=, you'll need to write some code to test your implementations. Here are the proper function declarations to add to "hashmap.h":

```
//
// copy constructor:
//
hashmap(const hashmap& other)
{
  //
  // TODO:
  //
}

//
// operator=
//
hashmap& operator=(const hashmap& other)
{
  //
  // TODO:
  //

  return *this;
}
```

Interesting side notes

There are two aspects of the program that you might find interesting. First, how is the **Hash()** function passed to the Insert() and Search() functions? In C, you would have to use function pointers, which are notoriously unsafe and error-prone. C++ once again takes a more modern approach and hides the pointers from you. Instead, we are working with objects of type **std::function**.

Open "hashmap.h" and look at the header declaration of **insert()**. You'll see the Hash parameter declared as following:

```
bool insert(KeyT key, ValueT value, function<int(KeyT,int)> Hash)
```

This declares **Hash** as a function of type *int(KeyT, int)*. This means the Hash() function must return an int, and take 2 paramters: a key and an int. Thinking in terms of the lottery program, look at the type of the actual Hash() function in "hash.cpp":

```
int Hash(string theDate, int N)
```

This matches the required type declaration, since the keys are dates --- i.e. strings.

Second, look at the Hash() function in "hash.cpp". Note that nearly all of the if statements are gone, and replaced by a **regular expression**. Regular expressions are a common technique for validation, and as you can see much simpler and cleaner. It requires learning a new language --- regular expressions --- but it's a very powerful technique. We'll discuss regular expressions in class on Monday April 6th; you'll also learn much more about REs in CS 301.

Requirements

1. The hashmap class must use hashing, and probing. You cannot use chaining of any type.
2. You cannot use any of the built-in data structures; you must continue to use the given dynamic array of structures. You may assume the hash table will never fill (i.e. that N was large enough when the hashmap was first constructed).
3. Add a copy constructor, and properly overload operator=.
4. You are required to free all memory allocated by your class. Valgrind will be used to confirm that memory has been properly freed.
5. No global variables.
6. If any of the requirements are broken, the grade is a 0.

Grading, electronic submission, and Gradescope

Submission: "hashmap.h".

Your score on this project is based on two factors: (1) correctness of hashmap.h" as determined by Gradescope, and (2) manual review of "hashmap.h" for commenting, style, and approach (e.g. adherence to requirements). Since this is part 1, it is worth 50 points for correctness, and 10 points for manual review of commenting and style. In this class we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness. Example: if you submit to Gradescope and your score is a reported as a 0, then that's your correctness score. The only way to raise your correctness score is to re-submit.

In this project, your "hashmap.h" will be allowed **12 free submissions**. After 12 submissions, each additional submission will cost 1 point. Example: suppose you score 50 after 15 submissions, and you activate this submission for grading. Your autograder score will be 47: 50 – 3 extra submissions. Note that you cannot use another student's account to test your work; this is considered academic misconduct because you have given your code to another student for submission on their account.

By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can **activate** an earlier submission if you want us to grade a different one; this must be done before the due date. We assume *every* submission on your Gradescope account is your own work; do not submit someone else's work for any reason, otherwise it will be considered academic misconduct.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%. After 24 hours, no submissions will be accepted.

All work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates"), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .