ICS 45C Spring 2019
Project #3: *Maps and Legends*

Due date and time: *Monday, May 27, 11:59pm*

Introduction

In the last couple of decades, two forces have combined to fundamentally change how software is built:

- The rise of an "always-on" Internet, local networks within organizations, and a growing variety of Internet-capable devices lead us to make assumptions about near-ubiquitous connectivity.
- The cost of acquiring and connecting many small servers has dropped precipitously. Thanks especially to cloud providers, it's no longer even necessary to purchase the servers; it's now possible to dynamically (and automatically) rent servers by the hour in data centers around the world.

What were once single, large-scale software systems running on beefy individual servers are now collections of cooperating services communicating with one another over networks. The paradigm of providing *software as a service* (i.e., providing the ability to access software running on the provider's servers — or even on servers rented by the provider from a third party — rather than providing software to be installed on the user's own infrastructure) is becoming increasingly popular and profitable.

As a result, it's become quite useful to break complex problems into collections of small services. The small services can be recombined more easily to solve new problems, and an organization may not even have to build all of the services it needs, instead subscribing to services offered by others to provide some of the necessary functionality. What appears outwardly to be a single web site might be composed, underneath the covers, of a variety of services provided by multiple distinct vendors.

In this project, we'll consider one such small service: an *authentication service* that manages usernames and passwords, but does little else. It will not be battle-ready — it'll store its information only in memory with no redundancy and will completely ignore security, for example — but it will serve as a vehicle for us to continue our recent exploration into writing *well-behaved* C++ classes and begin to consider the design of somewhat larger C++ programs, which will seed our work on future projects.

Well-behaved classes

We've been discussing in lecture what I call *well-behaved* C++ classes. A well-behaved C++ class is one whose objects can be used in ways that other types in the language can be used, while exhibiting the same kinds of behavior with respect to things like memory usage, parameter passing modes, and so on. Objects of well-behaved C++ classes clean up after themselves when they die, can be copied in a way that makes the copy entirely unique and separate from the original (or disallow copying altogether, when there is no reasonable way to provide that guarantee), can be made constant while preserving the ability to perform whatever operations do not change the publicly observable state of the object, and so on.

Every C++ class you write, starting with this project, will have to be a well-behaved class. What we'll discover as we go forward is that smart design choices — and the use of additional C++ features as we learn them — can make this goal simpler to achieve than you might think. But first we need to understand where the issues and pitfalls lie, and what tools C++ provides us to solve these problems, so we'll focus our energies on some of the lowest-level mechanisms first.

Getting started

As with the previous project, you'll need to complete a couple of chores before you can start working on this project. *Be sure you read these instructions and follow them completely before proceeding.*

Refreshing your ICS 45C VM environment

Even if you previously downloaded your ICS 45C VM, you may need to refresh its environment before proceeding with this project, so that you have a copy of the **project3** project template that you'll need for this project.

Log into your VM and issue the command **ics45c version** to see what version of the ICS 45C environment you currently have stored on your VM. Note, in particular, the timestamp; if you see a version with a timestamp older than the one listed below, you'll want to refresh your environment by running the command **ics45c refresh** to download the latest one before you proceed with this project.

```
2019-05-07 15:45:24
Project #3 template added
```

Note that you can instead use the **ics45c refresh_local** technique described in the Project #1 write-up, if you're unable to make your outgoing Internet connection work from within the ICS 45C VM.

Creating your project directory on your ICS 45C VM

A project template has been created specifically for this project. It includes a **gather** script for preparing your files for submission when you're finished, scripts for running the program with and without the Memcheck tool, scripts for running your unit tests with and without the Memcheck tool, as well as a sample input for your program (the same input that is specified in the project write-up) so you won't have to type it in repeatedly. (You will no doubt need additional test inputs besides the one we've provided, especially ones that contain fewer kinds of commands, so you can use them earlier in your development process.)

Decide on a name for your project directory, then issue the command **ics45c start PROJECT_NAME project3** to create your new project using the **project3** template.

*Do not use other project templates, like **basic** or **project2**, for this project!*

A brief tour of your project directory

After creating your project directory using the **project3** template, change into your project directory and issue the command **ls** to list its contents. Most of what you'll see is the same as in previous projects, but you'll see a couple of things that you'll want to take note of:

- The **inputs/** directory contains the sample input contained in this project write-up, which you can use for testing purposes — using the redirection trick shown in the previous project, you can avoid re-typing that input repeatedly.
- The **gtest/** directory, in which you'll write your unit tests, contains three files:
  - **gtestmain.cpp**, a file that provides a **main()** function for your unit tests; you will not need to modify this file.
  - **HashMapTests.cpp**, a file in which you'll write your unit tests.
  - **HashMap_SanityCheckTests.cpp**, a file that contains a set of "sanity checking" unit tests that will check that your HashMap class is compatible (i.e., it has the necessary member functions and you haven't introduced incompatible changes) with the tests we'll be running against it.
  - (More information about unit testing appears later in this write-up.)

The program

You will be writing an *authentication service*, whose role is to keep track of username/password combinations, verify that a particular username/password combination is valid, and be able to report on the number of unique username/password combinations that are currently known. As in the previous project, it will read all of its input from the standard input (**std::cin**) and will write all of its output to the standard output (**std::cout**), though you could certainly imagine it doing its work across a network connection instead. (If text-based communication like this seems primitive, you might be surprised to find out that many well-known Internet protocols actually send text-based commands and responses that are not all that different from what we're doing here, so we're in a more realistic scenario than you might think.)

Input and output

Your program should read one complete line of input at a time, parse it, and execute one *command*. Any command that is unable to be parsed correctly — because, for example, it has too few parameters or is a completely unrecognized command — should be seen as *invalid*. Valid commands, on the other hand, should be executed and will have some kind of observable effect, either changing what data is stored by the service or reporting back on some aspect of it.

The program continues reading, parsing, and processing one command at a time until a special "quit" command appears on the input, in which case the program ends.

More details about what the service does

The program stores a collection of username/password combinations in memory, arranging them in a data structure called a *hash table*. Initially, there are no username/password combinations stored; there are commands to create and remove them, as well as a command to verify that a particular username/password combination is correct, and a collection of *debug commands* that allow you to analyze the state of your hash table.

"Clean" memory usage

As in the previous project, before the program ends, any objects it has allocated dynamically must be deallocated. Furthermore, any illegal memory accesses — reads or writes to memory addresses that have

not been allocated (or have been deallocated already), use of uninitialized values — are prohibited, even if the behavior of the program is otherwise correct. (The latter rule is motivated by the fact that these kinds of illegal memory accesses exhibit behavior that is actually undefined in the C++ standard; that your program works correctly with illegal memory accesses when using one particular compiler on one particular operating system is no guarantee that it will work correctly on others, or even that it will always work on your own.)

As before, Memcheck is a great tool for verifying these conditions. If Memcheck reports no warnings as the program runs, and it reports no leaks when the program ends, you can rest assured that your program is using memory cleanly. You can run your program with Memcheck by issuing the command **./run --memcheck** from within your project directory.

The commands

There are two sets of commands that your service must provide. One is a set of *user-facing commands* that would presumably be used by subscribers to your authentication service; access to these commands is what you would hypothetically be selling if your service was available via the Internet. Separately, you'll provide a set of *debug commands*, which make visible certain aspects of the internals of your service (e.g., specifics on the structure of your hash table); these are commands you would not likely make available to a subscriber if this was a real Internet-based service, but might be very handy for finding problems when they occur.

The "user-facing" commands

The following user-facing commands must be supported. Every command appears on a line by itself, and the output of every command should appear on a line by itself.

Command Format	*Description*
CREATE *username password*	Create a new username/password combination and stores it in the program's collection. If successful, the output is **CREATED**. If the username is already stored in the collection, no change is made and the output is **EXISTS**. (One key consequence is that the same username cannot appear in the collection more than once.)
LOGIN *username password*	Checks a username/password combination to see if it is valid. A username/password combination is valid if it exists (i.e., the username is in the collection *and* is associated with the password), in which case the output is **SUCCEEDED**. If the username/password combination does not exist, the output is **FAILED**. (Note that there is no distinction between the username not existing and the password being incorrect; either situation would output **FAILED**. This is fairly standard — and wise — behavior for an authentication service; if you return different error messages when a username does not exist and when a password is wrong, you're letting an attacker know a useful piece of information: whether a username is valid.)
REMOVE *username*	Removes the username/password combination with the given username, if it exists. If so, the output is **REMOVED**. If no username/password combination with the given username exists, the output is **NONEXISTENT**.
CLEAR	Removes *all* username/password combinations from the service. The output is **CLEARED** (even if there were no username/password combinations at the time).
QUIT	The output of this command is **GOODBYE**. Once this command has been processed,

	the program should end.

The debug commands

The following debug commands must be supported. These are designed to provide you visibility into the internals of your service, which will assist you in your testing. They are also designed to provide *us* visibility when we grade your project.

Command Format	*Description*
DEBUG ON	Makes the other debug commands available. Before issuing this command, all other debug commands (except **DEBUG ON** and **DEBUG OFF**) should be considered invalid. If debug commands were not already on, the output is **ON NOW**; if they were, the output is **ON ALREADY**. While in debug mode, all of the user-facing commands remain available.
DEBUG OFF	Makes the other debug commands unavailable. After issuing this command, all debug commands should be considered invalid except for **DEBUG ON** and **DEBUG OFF**. If debug commands were already on, the output is **OFF NOW**; if they weren't already on, the output is **OFF ALREADY**.
LOGIN COUNT	The output is the number of username/password combinations currently being stored.
BUCKET COUNT	The output is the number of buckets in the hash table at present.
LOAD FACTOR	The output is the load factor of the hash table at present. There is no specific requirement here about the number of decimal places to include in the output; whatever emerges by default if you write a **double** to **std::cout** is best.
MAX BUCKET SIZE	The output is the length of the largest bucket (i.e., the one whose linked list contains the largest number of elements) in the hash table.

You can absolutely feel free to add more debug commands if you'd find them useful, though no additional debug commands are required. Debugging generally involves making things visible that are otherwise hidden. If you find yourself confronted with a scenario that you can't understand, think about what things you'd like to know that you can't already see, then add a debug command that makes it visible. (Examples might include being able to see which usernames are stored in which buckets, or being able to see the hash value of a particular username.)

Input and output requirements

All commands require all of the parameters listed above. The output for any invalid command — one that is missing parameters, has too many parameters, or is simply unrecognized (e.g., **LISTEN to music**) should be **INVALID**. (This time around, we will be testing your program with potential invalid inputs.)

Note that spelling is relevant here. You'll need to expect commands in the input to be spelled correctly, *and* you'll need to be sure that your output is spelled correctly (i.e., exactly as specified in the project write-up). When we grade your program, part of that work will be done using an automated test suite, so the spelling issue becomes vitally important.

Minor but important details

All input and output is case-sensitive; the word **BOO** would always be considered different from the word **boo** or the word **Boo**. (You'll find that this means you don't have to worry at all about this problem; string comparisons in C++, by default, behave this way.)

It is safe to assume that usernames and passwords can contain any character *other than* whitespace, but they can never have whitespace characters in them. (This simplifies the problem of parsing the commands.)

Anywhere that whitespace is permitted, any amount of whitespace is permitted. This means that the input commands below are all considered to be equivalent. (This, too, simplifies the problem of parsing the commands.)

```
CREATE boo@thornton.com sleeping
    CREATE      boo@thornton.com     sleeping
CREATE        boo@thornton.com sleeping
```

A complete example of program execution

The following is a complete example of program execution, demonstrating how input and output are interleaved. Input is shown in a regular font weight; output is shown in **bold**.

```
CREATE thornton@ics.uci.edu abcdefg
CREATED
CREATE boo@thornton.com sleeping
CREATED
CREATE boo@thornton.com playing
EXISTS
LOGIN thornton@ics.uci.edu abcdefg
SUCCEEDED
LOGIN thornton@ics.uci.edu defg
FAILED
LOGIN bill.gates@microsoft.com windows
FAILED
LOGIN COUNT
INVALID
DEBUG ON
ON NOW
LOGIN COUNT
2
DEBUG ON
ON ALREADY
DEBUG OFF
OFF NOW
LOGIN COUNT
INVALID
REMOVE thornton@ics.uci.edu
REMOVED
REMOVE thornton@ics.uci.edu
NONEXISTENT
REMOVE edge@u2.com
NONEXISTENT
```

```
LOGINS hello@hello.com hello
INVALID
LOGIN thornton@ics.uci.edu
INVALID
LOGIN
INVALID
WTF
INVALID

INVALID
QUIT
GOODBYE
```

Some background on our hash table implementation

Hash tables are implemented in many slightly different ways, but the central concept is always the same: When storing a collection of *search keys* (and possibly other information attached to each), define a way to determine where each search key "belongs," then use that as a starting point for deciding where to store the key and where to find it later. Deciding where a search key belongs is the role of a *hash function*, whose job is to take a key and return a *hash value*. The hash value is, in turn, used to choose a location to store, find, or remove the key, which can dramatically reduce the need to search for the keys (since you'll always know where to look).

Our hash table has the specific goal of acting as a *map*, which is a collection of key/value pairs; dictionaries in Python are maps, as are TreeMaps and HashMaps in Java. We'll implement our hash table in a class called HashMap. It will be *separately chained*, which is to say that it will be implemented as a dynamically-allocated array of *buckets*, where each bucket is a singly-linked list (or, more specifically, a dynamically-allocated array of pointers to nodes, with an empty list represented by **nullptr**). Because keys and values are paired together (i.e., each key has a separate value associated with it), each linked list node will store both a key and its associated value.

In a separately-chained hash table like ours, storing, removing, and looking up existing keys involves hashing the key, deciding which bucket it belongs in, and then searching that bucket by traversing that bucket's linked list.

Hash functions

Part of what makes a hash table work is the presence of a hash function. A hash function's job, ultimately, is to decide which bucket a particular key belongs in. Not surprisingly, there are lots of different ways to write a hash function, and a well-implemented hash table is not necessarily tied to a particular one; instead, it's ideally possible to configure different hash tables with different hash functions.

In our HashMap class, we'll solve this problem by writing two different constructors:

- One of them will take no parameters. A *default hash function* of your choosing will be used by any HashMap created using this constructor.
- Another of them will take a hash function as a parameter. Whatever hash function is given will be used by any HashMap created using this constructor.

Being able to configure different HashMaps with different hash functions will give us the ability to more thoroughly test your HashMap implementation (so that we can be sure we know how it will behave), and might also prove useful for you in our own testing.

Hash functions have the type **std::function<unsigned int(const std::string&)>**, which means they are functions (or anything else that can be treated as functions, which includes lambdas and objects with overloaded **()** operators) that take a **const std::string&** as a parameter and return an **unsigned int**. Note that these hash functions will have no other contextual information available to them; all they'll know is the string they're given when they're called. In particular, since they will be unaware of the number of buckets, they can actually legally return any arbitrary **unsigned int**, so it will be up to the HashMap class to take the values returned by the hash function and reduce them into the range of available bucket indices (e.g., by using the **%** operator).

Load factors and rehashing

While linked lists can grow with relative impunity — their size is ultimately limited only by the amount of available memory — the performance of a separately-chained hash table is a function of the lengths of its linked lists, so we're strongly incentivized to keep those lists as short as possible. Even with a wonderfully-designed hash function, a separately-chained hash table can still be slow simply because it's become overly full, with every list storing multiple keys. We'd like to avoid this problem.

This leads to a question: How do we measure how "full" a hash table is? We say that the *load factor* of a hash table is the number of keys it's storing divided by the number of buckets (or, stated differently, the average length of its lists). To avoid the performance hit of becoming overly full, your HashMap class is required to allocate a larger number of buckets and rehash all of the keys into them whenever the load factor exceeds a threshold of 0.8. (The reason that rehashing is necessary is that the number of buckets has an effect on which bucket a key will be stored in, so changing the number of buckets requires rehashing the keys so they're each stored in their new "home.")

Design requirements for your HashMap class

Your HashMap class *must* use the provided **HashMap.hpp** header file — which you'll find in your project directory after creating your project using the **project3** project template — as a starting point. That header file declares a set of members that your class is required to implement as-is — though you're welcome to add anything you'd like to it, you won't be able to change or remove *anything* — because we'll be running a set of unit tests against your HashMap class to verify its correctness, separately from the rest of your program. For example, if you change the signatures of the member functions declared in HashMap.hpp, our unit tests won't compile, and you will lose a substantial amount of credit on this project.

One of the primary goals of this project is to explore the tools provided by C++ to allow you to write a *well-behaved* class, so your HashMap class is required to be well-behaved (in the way we defined "well-behaved" in lecture).

Sanity-checking your HashMap implementation

To ensure that your HashMap is compatible with our unit tests, a set of "sanity-checking" unit tests are included in the **gtest** directory in your project directory, in a file called **HashMap_SanityCheckTests.cpp**. They make no attempt to validate any of the HashMap's functionality, but they do at least ensure that your HashMap contains all of the necessary public member functions, and that their parameter and return

types are correct. Initially, the sanity-checking unit tests will not link successfully — this why they're they are all commented out — but as you work, you'll be able to gradually uncomment them and see them compile and link successfully. If you haven't successfully uncommented, compiled, and linked all of the sanity-checking unit tests, your HashMap will not compile against our unit tests. (It's not a bad idea to run them, too; though they don't validate the functionality, they might expose a scenario where your HashMap crashes with a segmentation fault or something.)

As you work, you may discover that the sanity-checking tests that once compiled and linked successfully suddenly don't anymore; this is actually a clue that something important may have changed, so you'll want to be cognizant of it. Many of the errors you'll get from the sanity-checking tests are actually linker errors, which can be a bit difficult to unravel when you haven't had a lot of practice with them, but if you compile relatively often, there won't be many candidates whenever you have a problem; focus on changes you made most recently and you'll find your likely culprit.

Note that the sanity-checking unit tests won't tell you whether your implementations are correct, so you may still have issues even if they all pass. But you *certainly* have issues if they don't.

Some rules, limitations, and additional challenges

Here are the rules and limitations governing your work on this project.

- You are not permitted to use containers (e.g., **std::vector** or **std::list**) or generic algorithms (e.g., **std::find**) from the C++ standard library. We will be exploring the standard library in some depth in the relatively near future, but the goal here is to implement your own data structure by hand, to gain an understanding of how to build a well-behaved class out of underlying features that are not themselves well-behaved.
- Smart pointers such as **std::unique_ptr** and **std::shared_ptr** are off-limits here, since one of our goals is to learn more about manual memory management techniques.
- The public members of your HashMap class cannot be changed in any way from the way they've been declared in the provided **HashMap.hpp** header file — including seemingly minor changes, such as removing **const** from one of the member variable declarations. This is necessary so that we can compile and run our unit tests against your class, which will expect the public members to be identical to their current declarations. *If you introduce incompatible changes that cause our unit tests to fail, this will have a potentially significant impact on the score you receive on this project.* Your best bet is to make sure that all of the "sanity-checking" unit tests pass before you submit your work.
- Now that we're embracing C++'s object-oriented features, you should write classes other than just HashMap in your implementation. While there are no specific rules about precisely which classes you need, consider how you might slice the program's functionality into pieces or layers, representing each of those layers with a class. Classes provide the ability to combine data with operations, and that would certainly benefit parts of this project other than just the hash table.
- All of your classes should be well-behaved, no memory or resources should leak anywhere in your program, and memory should not be misused (e.g., storing a value in unallocated memory or derefencing a dangling pointer). Note, however, that classes whose member variables are all of well-behaved types are generally well-behaved without any extra work (e.g., you won't find that you need the Big Three in those classes). Don't write the Big Three unless you need them (e.g., don't include empty destructors in classes that don't require them, and don't write copy constructors or

assignment operators if all they do is the same thing that the default-generated ones do).
- Every class must be declared in its own header file and implemented in its own source file, with separation of interface and implementation as we've seen in the Notes and Examples thus far.

Additional challenges

As you work on the project, if you're interested in tackling additional challenges, here are a few directions you can go. In general, you should always feel free to explore the use of language features we've yet to cover, though you should also be aware that you sometimes won't be able to submit your work (if you choose features that explicitly violate one of the rules above, such as using C++ standard library containers like **std::list**), but that doesn't stop you from doing it as a learning experience, even if you don't submit it.

- One design challenge is to consider implementing your user interface using the Command pattern, with inheritance and polymorphism used to differentiate the different commands that can be entered via the standard input.
- Another design challenge is that the HashMap class is somewhat more limited than it could be, because it requires its keys and values to be strings. A more broadly useful HashMap would be implemented as a *template class*, meaning that individual HashMap objects can have their key and value types configured (e.g., HashMap<int, Student>). If you'd like to go this route, we can talk about ways to do it that won't break our unit tests; you'll have to approach it carefully, but it can be done. (Or you can work on that part separately and not submit it, which is probably a lot safer!)
- A useful optimization is to implement the ability for your HashMaps to be *moved*. You can accomplish this by adding a *move constructor* and a *move assignment operator* to your HashMap class, which requires the use of a relatively new C++11 feature called *rvalue references*. If you're interested in knowing more about the semantics of moving in C++ — which is beyond the scope of this course, but still good knowledge to have — check out the Move Semantics notes from my ICS 46 course.

We don't offer extra credit in this course, so things like this are strictly voluntary, but if you're looking for something interesting and additional to do, these are some ideas you might like to pursue.

Testing

Along with the program you'll be writing in this project, you will also need to write some *unit tests* that focus their efforts on verifying that your HashMap class is behaving as you'd expect. I'd actually suggest writing your HashMap class *and* its unit tests *before* writing any of the rest of the program, because it won't be possible to write your program without having a working HashMap implementation. (Of course, we'll be offering substantial partial credit to anyone who submits a working HashMap but none of the rest of the program, so there's additional incentive to work on the HashMap implementation first. We can test your HashMap without the rest of your program; we cannot test the rest of your program without your HashMap.)

When we grade your submission, we'll be running two kinds of tests:

- Whole-program tests, where we will redirect test input files into the program's standard input and check the output against our expectations
- Unit tests that focus only on your HashMap class, including member functions and other

functionality (e.g., the Big Three) that your program may not use

Why should we test functionality that's not used by the program?

You should think of classes as reusable components. To the extent that we can make their designs clear and their implementations bullet-proof, reuse will be enabled. For example, when you've finished your HashMap class, you should be able to write a second, separate program that uses it in ways that your original program didn't — and I'd suggest doing this in the course of your testing — yet still see, ultimately, that it works as it should; if not, you still have work to do.

How many tests do we have to write?

There is no explicit number of unit tests that we're requiring. Your goal is to write isolated, separate tests that together achieve coverage of all of the functionality in your HashMap class. Note that this is going to take more than just a small handful of tests, though each test is likely to be relatively short.

Writing your unit tests

You'll be using a unit testing library called **Google Test** to write your unit tests. We'll talk about Google Test in lecture, and that talk will be accompanied by a Notes and Examples section, but if you want to get a head start on reading about it, a good place to start are the documentation pages linked below:

- Google Test Primer — you'll definitely want to read this
- Google Test Advanced Guide — you may not find that you need this
- All of the Google Test Documentation

Write your unit tests in a file called **HashMapTests.cpp**, which you'll find in a directory called **gtest** inside your project directory. (Note that your unit tests do not belong in the **app** directory, because they are not part of your program; they're separate.) I've already created that file for you, and I've even given you one unit test already implemented, though it's commented out initially, since it depends on functionality you've not yet written. Once you get the necessary member functions in your HashMap class implemented, you can feel free to use the test I provided as one of your unit tests.

Running your unit tests

As with previous project templates, when you compile and link your code using the **./build** script, there are actually three separate executables being built:

- **app**, which is your program (which starts executing in **main.cpp**)
- **exp**, which is any experimental code you want to write (which starts executing in **expmain.cpp**)
- **gtest**, which is your unit tests

To run your program, you can use the **./run** script as usual. To run either of the others (**exp** or **gtest**), you can use **./run exp** or **./run gtest** instead. Note, too, that you can run your unit tests using Memcheck with the command **./run --memcheck gtest**, too, if you're so inclined, and that you can build just one executable with a command like **./build gtest**.

You can try all of this out before you write any code. Simply run the **./build** script and then issue the command **./run gtest**, which will show you output that looks something like this:

```
[==========] Running 0 tests from 0 test cases.
[==========] 0 tests from 0 test cases ran. (1 ms total)
[  PASSED  ] 0 tests.
```

Submitting your unit tests

The **gather** script will gather files in your **gtest** directory, along with the files in your **app** directory, so your unit tests will automatically be included in your submission.

Deliverables

After using the **gather** script in your project directory to gather up your C++ source and header files (including your unit tests) into a single **project3.tar.gz** file, submit that file (and only that file!) to Checkmate.

Follow this link for a discussion of how to submit your project via Checkmate. Be aware that I'll be holding you to all of the rules specified in that document, including the one that says that you're responsible for submitting the version of the project that you want graded. We won't regrade a project simply because you submitted the wrong version accidentally. (It's not a bad idea to look at the contents of your tarball on your host operating system before submitting it.)

Can I submit after the deadline?

Yes, it is possible, subject to the late work policy for this course which is described in the section titled *Late work* at this link.

Several tweaks and adjustments by Alex Thornton, Fall 2016.
A handful of tweaks by Alex Thornton, Fall 2015.
Additional tweaks for new ICS 45C VM by Alex Thornton, Fall 2014.
A unit testing requirement, additional tweaks, and new explanations added by Alex Thornton, Winter 2014.
Parts rewritten and additional debugging commands and information added by Alex Thornton, Fall 2013.
Originally written by Alex Thornton, Fall 2012.