**Link to Code:**

[**https://replit.com/@RacheySnachey/c11hwstartpotions#main.cpp**](https://replit.com/@RacheySnachey/c11hwstartpotions#main.cpp)

# **Goal**

As a new developer at a mobile game company, you have been asked to create a Harry Potter "Potions" Game. Your game will allow a Wizard (the player) to discover new **Potions** by brewing fundamental **Ingredients** in various combinations and quantities using a **Cauldron**, a **Pantry**, and a **PotionsBook**. When the Wizard discovers a new Potion through Trial-and-Error brewing, new ingredients will be created and added to the Wizard's Pantry. (But only if you already have the necessary ingredients in your pantry to brew your newly discovered potion!)

For this assignment, you will only need to finish implementation of two classes: Potion and PotionsBook, but they must be implemented according to the prototoypes provided in the associated .h files for each class.

* The Wizard (player) starts out the game with a bare minimum of fundamental ingredients in the Wizard's **Pantry**, along with a **PotionsBook** that has all of its Potions magically hidden in invisible ink. The Wizard will need to experiment with different combinations of available ingredients found in her Pantry to discover and reveal new potions, and to create new ingredients using those potions.
* The Wizard will need to place random collections of available ingrediens into the Cauldron, in varying amounts, to determine if they can be brewed, thus revealing a new Potion in the PotionsBook.
* Only specific combinations and quantities of available ingredients currently found in the Wizard's Pantry will produce a successful brew (and thus generate new ingredients)
* When a Wizard successfully brews (discovers) a Potion, the ingredients that had been placed in the Cauldron will be "used up" by the Potion, and thus removed from the Wizard's Pantry. (If the exact quantity and type of ingredients in the Cauldron are not available in Wizard's Pantry, the brew will fail, and the Pantry will be unaffected. In this case, even a valid Potion (if still hidden) will remain undiscovered.)
* Upon successful brewing, one or more new ingredients will be produced by the Potion (each in specific quantities), and then those new ingredients will be placed in the Wizard's Pantry, leaving the Cauldron empty. The type and quantity of ingredients produced by a successful brew will depend on the specific Potion that was discovered (some Potions will produce 1 unit of 1 type of ingredeient, while other Potions may produce multiple ingredients in multiple quantities)
* If the Potion "just brewed" had never been brewed before (that is, it was just discovered), the new Potion will be magically revealed in the PotionsBook, for future reference. You can "inspect" the PotionsBook at any time by printing its contents. Only "discovered" potions - or Potions which were initially granted as "visible" at the start of the game will be printed.
* The goal of the game is to discover (reveal) all possible Potions in the PotionsBook and to build up a formidable Pantry of exotic ingredients in large quantities.

# **Ingredients**

* Ingredients in the game will be represented by std::string names, like "air", "water", fire", etc. (There is no Ingredient class)
* Each ingredient name should be unique among all other ingredients in that are used in working Potions. To keep things simple, ingredient names shoud NOT contain spaces. But they can contain capital or lower case letters and numbers. However (!!), remember that "water" (lowercase w) would be different than "Water" (capital W). So name your ingredients wisely to avoid user error (perhaps all lower case?) Or better yet change all user input to lower case to avoid any confusion if the user accidentially uses upper case.
* Ingredient quantities that are available in the Pantry or which are required to make a specific Potion will always be represented by integer values. For example: steam = 1 unit of fire + 2 units of water, etc. (not 1.5 units of water)

# **The Wizard's Pantry**

* The Wizard will keep her available ingredients in a Pantry which will be represented by a std::map<string, int> container, where each ingredient is a key:value pair, represented by std::pair.
* See <https://en.cppreference.com/w/cpp/utility/pair> for more information on the pair class)
* The "key" of each ingredient std::pair in the Pantry will be the string name of the ingredient (ex: "fire"), and the "value" of each std::pair will be the integer quantity of that ingredient currently available in the Pantry(ex: 4). Recall that std::pair will allow access to its 2 members via the public member variables first and second. Thus, any particular std::pair from the std::map<string, int> will hold the ingredient's string name in first and the ingredient's quanitity in second.
* Note that "first" and "second" are data members of std:pair, NOT METHODS (so we do NOT use parenthesis with "first" or "second" when accessing the std::pair members).
* The Pantry does not need a class representation, because the std::map<string, int> will provide all the functionality we need to search and update the Pantry when brewing. As such, you simply need to create an instance of an std::map<string, int> container to represent the Pantry in your main(), and give it a "variable name" so that you can iterate through its contents and inspect them, and update them, using the std::pair members first and second.
* As the developer, you will need to place some initial ingredients in the Pantry so the Wizard can begin brewing and producing more, new ingredients in the Pantry. Consult the pantryStart.txt file for a list of ingredients and quantities you should place in the Wizard's Pantry at the start of the game. You should feel free to create as many new ingredients as you like in the Wizard's Pantry to start the game (and any corresponding "undiscovered" Potions that may use them).
* You will add initial key:value pairs of <string, int> to your Pantry in main(), simply by assigning a value to a specific key in the map, as indicated by the following code:

// (Note: scroll/drag right to see the entire code comment section below)

std::map<string, int> myPantry; // create an empty pantry called myPantry with a std::string key, and an integer value

myPantry["fire"] = 3; // 3 units of fire are now in the myPantry map

if (myPantry.count("fire")>0) // ONLY if there is a key in the map called fire....

cout << myPantry["fire"] << endl; // ... then print out the number of units of fire in the map

# **The Cauldron**

* The Cauldron is where the Wizard will try to mix an arbitrary combination and quantity of different ingredients, to make some kind of new potion.
* The Wizard can put any number of ingredients into the Cauldron simply by invoking their name. In the game, this will be done by typing any string name on a single input line and hitting return. The named ingredient will be immediately placed in the Cauldron, regardless of whether it exists in the pantry, and even if the name is incorrectly typed, or if it is not a valid ingredient in the game.
* The Wizard can increase the quantity of an ingredient in the Cauldron simply by entering the ingredient name multiple times. (This should make user input a bit easier to process, but you are welcome to modify your user input interface to enable more powerful expression of ingredient names and quantities)
* The ingredients named by the Wizard which have been placed in the Cauldron may or may not exist in the Wizard's pantry and may or may not comprise a viable Potion. The only way to know if the ingredients that have already been placed in the Cauldron comprise an actual Potion is to attempt to brew them.
* If the Cauldron's ingredients exist in the Pantry in the same quantities as found in the Cauldron, AND if specific combinatation of ingredients in the Cauldron matches any hidden or already-discovered Potion in the PotionsBook, only then will the Potion be successfully brewed and revealed in the printed PotionsBook (if not already known).
* The Cauldron is very much like a temporary Pantry. That is, it will be represented by a std:map<string, int> container of std::pair where the first member of the std::pair (the "key" of the map container) will be a std::string, and the second member of the std::pair (the "value" of the map container) will be an integer. The Cauldron thus represents what the Wizard has requested to be "Brewed" as some new potion. The Caudron ONLY represents the Wizard's requested ingredients and quantities; there is no guarantee that the Wizard's Pantry will contain the requested ingredients in the quantities the Wizard has requested. There is also no guarantee that the contents of the Cauldron represents a viable Potion.
* If the Pantry contains the exact type and sufficient quantity (equal, or more) of the ingredients in the cauldron, then brewing may be attempted. As the developer, you will need to check the Wizard's pantry for each ingredient found in the cauldron, in the exact quantity, and abort the brewing attempt, if they are not found, or if the quantity of ingredients requested is larger than the amount on hand (hint: this refers to the "value" portion of our Pantry std::map.)
* See the section on Brewing below, to learn more about what happens when the Cauldron's ingredients ARE available in the Pantry.

# **A Potion**

A Potion will be represented by a C++ class and it is essentially a recipe, with the following data members:

* A boolean value wasDiscovered that indicates if this Potion has already been discovered by the Wizard. Until it is discovered, every Potion in the PotionsBook is "hidden" and undiscovered. A Potion can only be discovered by a Wizard through a Trial-and-Error process of combining different ingredients in different quantities in a Cauldron. Note: a few Potions may be "discovered" already at the beginning of the game to help a new Wizard get started.
* A Potion has a std::string name that describes roughly what the potion produces or what the Potion does: "steam", "fog", "water-duplication". This is a descriptive name and is not necessarily the same as the specific ingredient(s) produced by the Potion. For example, a Potion called "water-duplication" will have a string name "water-duplication", but it will produce 2 units of the ingredient "water", based on a Cauldron that only contains 1 unit of the ingredient "water" (thus increasing total units of water in the Pantry).
* For simple Potions, the name of a Potion may match the name of the primary ingredient created by the Potion. For example, a Potion that creates steam by combining water and fire can be called "steam", but the ingredient it creates must be specified separately as "steam". This will likely be a point of confusion for some developers as they get to know the Potion class.
* ingredientsRequired A Potion has a collection of ingredients that are required to make it, in specific quantities: "water:3, fire:2, earth:4" etc. This "collection" of required ingredients will actually be represented by a data member which is of type std::map<string,int>. As with the Pantry and the Cauldron, this list is a collection of std::pair instances that represent the key ("ingredient" string name) and value (integer quantity) for each ingredient required to make the potion. It should be noted that you can trivially compare the contents of the Cauldron (a std:map<string,int>) with the collection of required ingredients in any Potion, to see if they are equal, simply by using a == comparison operator. If they are equal, then the Cauldron has the exact combinataion and quantities of ingredients to make that specific potion. This comparison is trivial and you do not need to iterate through the entire map to assess whether or not the Cauldron matches a particular Potion's collection of required ingredients.
* Please note that although two potions like "volcano" and "stone" are both comprised of "fire" and "earth", the "volcano" potion requires 2 units of "earth" while the "stone" potion only requires 1 unit of "earth". (so the map that represents their required ingredients are not equal)
* ingredientsProduced A Potion also has a collection of ingredients that are produced when the potion is Brewed. Note that although a Potion will often produce only a single ingredient ("water + fire = steam"), a Potion can also produce MULTIPLE DIFFERENT INGREDIENTS in DIFFERENT quantities. So for example, a "water-duplication" potion could REQUIRE 1 unit of the ingredient "water", and it would PRODUCE 2 units of "water" as a result of brewing the Potion (helping you increase your supply of the water ingredient).
* Similarly, any Potion, when successfully brewed, could also produce "experience points" that will also be placed in the Wizard's Pantry along wth any newly produced Ingredient, as a way of tracking a Wizard's overall mastery of all Potions. (but this is an idea for an ehnancement, and not a requirement of this assignment)
* Thus, the std:map<string, int> of ingredientsProduced by the Potion could specify only one key:value pair (an ingredient and its quantity), or it could alternatively specify multiple ingredients in various quantities produced by the Potion (many key:value pairs)
* Like the Pantry, and the Cauldron, the "collection" of ingredient(s) produced by the Potion will be a std::map<string,int> comprised of a collection of std::pair instances that represent the key ("ingredient" string name) and value (integer quantity) for each ingredient produced by the potion.
* Again, very often, this "collection" of what is produced will only have one ingredient and a quantity, (and that ingredient's name may match the "Potion name", but not always).

# **Brewing a Potion**

* If the Wizard comes up with a combination of ingredients in the Cauldron that is found to match either a discovered or previously unknown Potion from the PotionsBook, only then can the brewing process commence. If the Cauldron does not match any hidden or discovered Potion in the PotionsBook, then the brew should be aborted, and the Cauldron should be "emptied" (and the Pantry will be unaffected).
* When the Cauldron does contain the exact combination of ingredients found in a Potion in the PotionsBook, AND the Wizard's Pantry contains all of those ingredients in sufficient quantities, then Brewing is in fact relatively straightforward:

1. The contents of the Cauldon must first be subtracted from the Pantry. (If the contents of the Cauldron do not exist in same or greater quantities in the Pantry, then the brewing must be aborted, and the Potion, if valid and still hidden, must not be revealed ).
2. After the correct number of ingredients are subtracted from the Pantry, the Potion being brewed must be consulted and the correct number of each Ingredient produced by the Potion must be added to the Wizard's Pantry. If the Wizard's Pantry already contains one or more of those Ingredients, then their existing quantities must be incremented. If the Wizard's Pantry does not already contain those ingredients, then a new "key:value" std::pair will need to be added to the Pantry, which is represented by the std::map<string, int> container.
3. After successfully brewing a previously undiscovered Potion, that specific Potion in the PotionsBook must be "discovered" by setting the boolean value of its wasDiscovered data member to true. Note that only "discovered" Potions will be listed when the PotionsBook is printed, but as the developer of the game, you will of course know exactly what all the Potions are because they will be stored and "hidden" in the Potions book when the game starts. (except for Potions you choose to reveal at the start of the game)

* An attempted brew fails either because the contents of the Cauldron are not available in the Pantry in the exact quantities found in the Cauldron, or it fails when the contents of the Cauldron do not exactly match any hidden or already-discovered Potion's required ingredents, as found in the PotionsBook. When a brew does fail, the contents of the Cauldron are considered "emptied" and there is no affect on the Pantry.
* Your program should never subtract ingredients from the Pantry before a successful brew occurs. The Cauldron represents a "transient magical state" where Ingredients which "may or may not exist in the Pantry" can be collected. Only through the process of attempting to brew the contents of the cauldron will we find if the Ingredients actually exist in the pantry and if they match a viable Potion in the PotionsBook.

# **The PotionsBook**

* The PotionsBook is represented by a C++ class. It is essentially a collection of instances of the Potion Class (see Potion, described above). A data member in the PostionsBook class called PotionsBook::potions is a std::map comprised of std::pair instances. Each std::pair is a standard key:value pairing of the key (potion name string) and value (Potion class instance). The class provides a few basic methods for helping to test, manage, and print PotionsBook entries.
* The Wizard can request to print the contents of the PotionsBook at any time, but only "discovered" Potions will be printed during the game. Invisible (undiscovered) potions will remain hidden in the PostionsBook until they are discovered through brewing. (The "hidden" state of each potion is actually a data member in the Potion class, not the PotionsBook).
* At the start of the Game, the PotionsBook will appear empty to the Wizard when printed, but in fact it will contain all possible potions, which will be "magically hidden" by the boolean wasDiscovered value in each Potion instance. (Note: you can choose to reveal key potions to help the player get started.)
* The PotionsBook data member "potions" will actually be a std::map<string, Potion> container which will provide immediate access to a Potion instance when looked up by name, or by iterating through the entire this->potions map on Potion at a time to print each "discovered" Potion.
* Remember that the std:pair will be used to hold the key and value of the each entry in the PotionsBook::potions data member. This means that the first member of the std::pair will be the descriptive name of the Potion, and the second member of the std::pair will be the actual INSTANCE of the Potion. Thus if we have a variabled calls thisEntry that contains an instance of a std::pair in PotionsBook::potions, then we can tell if that specific Potion has been discovered or not by using this statement: if ( thisEntry.second.wasDiscovered) // we can print this potion.. This is because the second entry is the instance of a Potion, and the Potion instance has a member named wasDiscovered.
* Hidden Potions will remain hidden until that specific Potion has been "discovered" by the Wizard through brewing.
* The Wizard will only be able to see a Potion after discovering it through Trial-and-Error brewing with the Cauldron. This means that attempting to brew the contents of the Cauldron (assuming the Pantry contains all the ingredients in the Cauldron) requires your program to search through all Potions in the PotionsBook (whether they are discovered or not), checking to see if any Potion's required ingredients exactly matches the current contents of the Cauldron. If any Potion in the PotionsBook has the exact collection of ingredients in the Cauldron, then that Potion will be brewed (assuming the Pantry has all the necessary ingredients in sufficient quantities).
* Even a Cauldron that has an exact match of required Ingredients for a Potion in the PotionsBook cannot be brewed unless all of the required Ingredients exist in the Wizard's Pantry in the correct amounts.
* When a Potion from the PotionsBook is brewed for the first time, it will be "discovered" and thereafter it will show up when the Wizard tries to print the PotionsBook for reference.
* See the section on Brewing a Potion to understand how successful Brewing affects the Pantry.

# **Main**

* main() has been partially written for you, but you will need to fill in its missing logic
* You are free to change main() as long as it satisfies the requirements of the assignment described herein.
* main() will be where you create and initialize the Wizard's Pantry with a simple std::map :

map<string, int> myPantry = { {"fire", 1}, {"water", 1}, {"air", 1} }; // collection of our initial ingredients

* you will also create an instance of the PotionsBook class:

PotionsBook myPotions; // empty book of potions - an instande of the PotionsBook class

* You can create a new Potion like this

Potion myPotion( "steam", { {"fire", 1}, {"water", 1}}, { {"steam", 1} } ) ); // create an instance of a potion using the Potion constructor

* And then you can add potions to the class like this:

myPotions.addPotion( myPotion ); // pass a copy of a new potion to the addPotion() method

* Alternatively, you can add potions directly to the PotionsBook instance like this:

myPotions.addPotion( Potion( "steam", { {"fire", 1}, {"water", 1}}, { {"steam", 1} } ) ); // pass a copy of a new potion to the addPotion() method

* Additinally, you can add potions using curly-braces like this (because addPotion() 'knows what it needs')

myPotions.addPotion( { "steam", { {"fire", 1}, {"water", 1}}, { {"steam", 1} } } ); // pass a copy of a new potion to the addPotion() method

* All of the above approaches are valid ways to add a std::pair to the instance of PotionsBook called myPotions.
* Please note that the PotionsBook is not a map. It is an instance of a class that contains a data member called "postions" which itself is a map. The PotionsBook class also has methods.

# **Testing with PotionsBook::selfTest()**

* When you are ready, you should be able to test your potions book in main() as shown below, by using the already-written method PotionsBook::selfTest(). This method will 'exercise' each of your Potions found in the book by trying to create a "perfect cauldron" of ingredients and then calling your "brew" method to see if it updates the pantry and the PotionsBook accordingly. Note that if your PotionsBook is empty, you will not really be testing your methods sufficiently - you will want to fill your PotionsBook before making this call

myPotions.selfTest(); // call the selfTest() method of the PotionsBook class on a "full" book of potions (an instance of PotionsBook)

# **Sample Output**

Welcome Potion Class!

Your pantry currently contains 3 different Ingredients:

air (1)

fire (1)

water (1)

Your PotionsBook currently contains 3 total potions.

Only the following potions have been discovered so far:

Potion: fog requires ingredients: air:1 + water:1, and produces: fog:1

Enter the letter 'h' (then return), for help, or enter any command: b

Brew a Potion....

Enter the name of an ingredient to add to your cauldron (or x to stop adding): air

Enter the name of an ingredient to add to your cauldron (or x to stop adding): water

Enter the name of an ingredient to add to your cauldron (or x to stop adding): x

Searching for a potion that matches your Cauldron's contents...

I found a potion called fog that you can make with those ingredients!

... checking your pantry to see if you have enough of each ingredient...

Brewing fog...

Success: Just brewed 1 units of fog and placed in your pantry (removed 1 air and 1 water from the pantry)

Done Brewing Potion "fog"

Enter the letter 'h' (then return), for help, or enter any command: l

Your Pantry contains the following items:

air (0)

fire (1)

fog (1)

water (0)

Enter the letter 'h' (then return), for help, or enter any command: p

Your PotionsBook currently contains 3 total potions.

Only the following Potions have been discovered so far:

Potion: fog requires ingredients: air:1 + water:1, and produces: fog:1

Enter the letter 'h' (then return), for help, or enter any command: b

Brew a Potion....

Enter the name of an ingredient to add to your cauldron (or x to stop adding): fire

Enter the name of an ingredient to add to your cauldron (or x to stop adding): water

Enter the name of an ingredient to add to your cauldron (or x to stop adding): x

Searching for a potion that matches your Cauldron's contents...

I found a potion called steam that you can make with those ingredients!

... checking your pantry to see if you have enough of each ingredient...

Sorry.. you do not have enough water to brew that Potion.

Brewing Aborted. Snape is not pleased.

Sorry no brewing could occur with your current Cauldron Contents

Enter the letter 'h' (then return), for help, or enter any command:

# **Getting Started**

You may want to follow the suggested implementation order below, but it is not required:

1. Create a map<string, int> myPantry variable in main(), and assign it key:values from the pantryStart.txt file. This map variable will represent your Wizard's Pantry (no Pantry class is needed)

2. Print out the contents of your pantry variable using a range-based for loop, and the "auto" keyword:

for( int x : manyXs ) { // loop one time for each item in manyXs (`auto` can be used instead of 'int' here)

cout << x.first << ", " x.second; \\* do something with x; remember it is a std::pair \*\

}

1. Create a loop in main() that promps the user for a command; tell the user 'h' will print the help instructions
2. Inside your loop in main() use a switch statement to determine which 1st character was entered by the user
3. Within your switch statement, create 4 cases: (l) print (list) the contents in the pantry, (p) to print the PotionsBook, (h) to print this list, (b) to start brewing a potion, and (q) to quit the game, and default in case a bad user entry is made. Note that you will not yet be able to take action on all users requests, except printing the Pantry contents.
4. Test your program to be sure you can enter all possible switch cases and print an appropriate test response for each case (for now). Be certin your (l) case does print the Pantry's current ingredients (and the amount of each ingredient).
5. Implement your Potion class constructor and just the printPotion() method (for now). Use the sample output shown above to guide your formatting for your printPotion() method. (it does not need to be exact but all aspects of the potion should be presented EXCEPT the wasDiscovered member). Your output for a specific potion should look like this: Potion: fog requires ingredients: air:1 + water:1, and produces: fog:1
6. In main() before your loop, create instances of a few individual "test" potions using potionsStart.txt as your guide. Create Potion instances in main() as individual variables for now, and try to print them out to convince yourself that your Potion constructor and your printPotion() method are working correctly. You will delete these Potion instances after you are sure they are working, because we will eventually put all Potion instances in the PotionsBook.
7. Implement your PotionsBook class constructor and its methods addPotion() and printBook().
8. In main(), test your PotionsBook class by creating a PotionsBook instance in main() called myPotions, and calling the myPotions.addPotion() method to add multiple potions from as found in the potionsStart.txt file (you do not need read the file with your code, just use it as reference).
9. Test your myPotions.printBook() method to be sure that it will only print DISCOVERED potions by default, and that if you pass a boolean "true" value as the argument to the printBook() method, it will be forced to print all Potion instances in the book.
10. Return to your Potion class and implement the Potion :: canBrew(const map<string, int> &pantry ) method. This method will only return true if the pantry reference argument provided contains at least enough of the all required ingredients needed to brew the Potion. Your canBrew() method can access the Potion data members this->ingredientsRequired directly and compare the key:value of each member of the ingredients required with the current quantity of "onhand" ingredients in the Pantry reference provided. Your canBrew() method will NOT be allowed make any changes to the pantry. That is why we may it a const type argument. (We make it a reference argument for performance reasons... so we do not have to make a copy of the entire Pantry just to pass it to the canBrew() method. Const and references are often used together in this way!)
11. Continue implementing all remaining methods in your cpp files to match their .h prototypes.

# **Implementation Constraints and Scoring**

* Output from the selfTest() method of the PotionsBook class will be used to determine scoring on the assignment
* More details to be provided toward the end of the semester
* 100 points total

# **Bonus (+15 points)**

* Implement the 2nd Constructor for the PotionsBook class to read the provided potionsBookStart.txt file directly into the PotionsBook data member called potions. (Once working, test your new constructor on the much larger potionsBookPro.txt file)
* Updated main to utilize your constructor to load the file directly, and then allow the selfTest() method to be run on this copy of your PotionsBook before starting the game.
* The file format is layed out as follows (comments below after // are NOT in the actual file)

water-duplication // 1 word: descriptive name of this potion - one word only, no white space

wasDiscovered // 1 word: a hint - if the word == "wasDiscovered" then we need to set: Potion::wasDiscovered = true

ingredientsRequired 1 // 2 words: "ingredientsRequired" can be read/thrown away;

// .. we want the integer that indicate how many rows of required ingredients follow this line

water 1 // 2 words: name of required ingredient and integer quantity of that ingredient. These become a pair

ingredientsProduced 1 // 2 words: "ingredientsProduced" can be read/thrown away;

// .. we want the integer that indicates how many rows of produced ingredients follow this line

water 2 // 2 words: name of produced ingredient and integer quantity of that ingredient. These become a pair

air-duplication // next ingredient repeats...

wasDiscovered

ingredientsRequired 1

air 1

ingredientsProduced 1

air 2

# **Possible Enhancements**

* If you complete the assignment including the Bonus and would like to extend your "crafting engine", here are a few ideas you may find worthwhile. Consider also that what you have created is a common module in many many crafting games. By changing its content (potions and prompts) you can create a wide range of different games. (This assignment can take you very far toward building a personal project that you can share with prospective interviewers for internships and job applications, especially in the Games Industry - but it could be easily adapted to a different application as well)

1. Experience: Add an "xp" or experience "ingredient" that is produced by each type of potion, so that as a wizard you can accrue more experience
2. Save PotionsBook Progress: Incorporate a new file type to your game that you read and write that will keep track of which potions have been discovered and which have not, so that you can continue your progres between game sessions.
3. Save Pantry: Similar to above: incorporate a new file type to your game that you read and write that will keep track of the ingredients you currently have in your pantry.
4. Currency: Add a currency system that will allow you to purchase items: unlock a potion, or purchase a crucial ingredient. Perhaps sell an rare ingredient back to the "bank" for money in return. Keep track of your currency simply as another ingredient in your pantry (currency name and quantity). Consider supporting multiple currencies - a premium currency and a "grind" currency, as is done in many mobile games.
5. Delay Mechanic: Add a time-delay mechanic to your brewing, so that every potion takes a prescribed number of seconds to brew. More sophisticated potions may take longer to brew - requireing the user to "come back later" to see when their potion is ready to "store" in the pantry. This is a common "re-engagement" mechanism used in many games. Hint: You may need to incorporate use of "threads" as discussed in class to enable additional brewing in a separate Cauldron, while one of your Cauldron's is busy brewing a potion.  
   6: Upgraded Cauldrons: Consider a urchase (or a potion) that enables a Cauldron to be upgraded so that it might brew faster. Or so that it can brew "advanced potions". Or perhaps only allow a Cauldron to be upgraded after the user has earned a number of experience points.
6. Multiple "Crafting" Devices: Consider supporting more than "crafting" device (in addition to Cauldrons). Perhaps an Oven or a Grinder. Each of these devices would only support "crafting" a specific type of potion. This would require extensions to your Potion class and data files to support an additional field that indicates the name of the device that is needed to "craft" the potion. Almost like a new type of required ingredient, but one which is not used up when done (but it stays "busy" during crafting). Consider that these crafting devices expand the potential for you game significantly cover game types like castle-defense (Clash of Clans) and Farming (Hay Day).

# **Concepts Covered in this Homework**

* std::map container
* brace-initialization of std::map containers
* backet-initialization of std::map key:value pairs
* testing for existence of std::map keys (BEFORE checking their value!!!)
* loops: while, for
* compound logic conditionals
* pass-by-value vs pass-by-reference
* user input and output via cout and cin
* range-based for-loops and the "auto" keyword
* references
* Advanced C++ class concepts: instances, constructors, methods,
* Accessing C++ class members of classes via "dot" or "arrow"
* Separating C++ class prototypes (.h file) from implementations (.cpp files)
* Using "include guards" in .h files to ensure only one prototype per compilation unit