Programming Assignment 4
Pirates on the High C's - Episode 2: There Be Krakens
Inheritance, Standard Template Library (STL), Exceptions, and Persistent Objects

Out: November 23, 2015, Monday -- DUE: December 10th, Thursday, 11:59pm
EC327 Introduction to Software Engineering – Fall 2015

Total: 200 points

- *You may use any development environment you wish, as long as it is ANSI C++ compatible. Please make sure your code compiles and runs properly under the Linux/Unix environment on the PHO 305/307 (or eng-grid) machines before submitting.*
- *You will receive a 0 if your code does not compile on the PHO 305/307 machines.*
- ***NO LATE PA4s will be accepted.***
- *Follow the assignment submission guidelines in this document **AND** match the provided sample output document **exactly** or you will lose points.*

Submission Format (Must Read)

- Use the exact file names specified in each section for your solutions.
- Complete submissions should have **28 files (see the end of the document for the complete list).** Put all of your files in a single folder named: ***<your username>_PA4*** (e.g., dougd_PA4), zip it, and submit it as a single file (e.g., dougd_PA4.zip).
- Submit your .zip to the PA4 portal on Blackboard by 11:59pm on the due date.
- List in the body of your README.txt the steps you completed. For example just say: Step 1, Step 2, Step 3 etc. Feel free to provide additional CLEAR and CONCISE additional explanations. We want to understand what you think works and what doesn't. This will help with grading. Be honest.
- Please do **NOT** submit *.exe and *.o or any other files that are not required by the problem.
- **Code must compile in order to be graded.**
- Comment your code (good practice!) We **may** use your comments when grading.

Overview

Note: This project requires a working version of programming assignment 3 (PA3). Please work with the staff to make sure this happens as quickly as possible.

Many of the implementation details in this assignment are under your control, meaning you will have to do more design work on this assignment than previous ones. Most people find that designing code is a lot harder than writing it, especially in the middle of the night. So start the assignment early enough to be able to spend some primetime on it.

The assignment is presented as a recommended series of steps that start with working version of PA3.

Step 2 requires Step 1, and consists of integrating the Step 1 STL "list" class into PA4. Step 5 should not be done prior to Step 2 to avoid wasting coding time. Steps 3 and 4 as a pair can be done at any time, and likewise for Step 6. The specified order is recommended as the most efficient and logical.

"There Be Krakens" Instructions

The goal of "There Be Krakens" is to establish a pirate-infested sea where the objects interact in new ways. Also the game now can be saved and restored and has robust error checking.

"There Be Krakens" consists of five key game objects: docks, ports, merchants, pirates, and one Kraken.

- **Docks** have a certain amount of space to provide home and shelter for merchants.
- Merchants and pirates are **sailors**. Sailors live at docks, sail around, and gather supplies while consuming their cargo, which then shrinks in size. When their health is depleted, the sailors die.
    - **Merchants** can only gather supplies at ports, helping their cargo to grow in size. Additionally, ports will create new merchants and pirates in a deterministic fashion, expanded below.
    - **Pirates**, on the other hand, attack merchants. Also, pirates can attack each other. If a pirate and a merchant are in the same dock, then the pirate kills the merchant. If the pirate attacks a merchant, the pirate takes part of the merchant's cargo.
- **Ports** represent supply resources only for merchants.
- **Krakens** appear **every 15** time ticks. Krakens churn up the pirate-infested sea and sweep away all living merchants that are not at a dock.

The player enters game commands through a simple command line interface in order to create new game objects and to instruct sailors to perform actions. The number of possible game objects is limited though. Entering a new command increments the Kraken clock for one tick. At every tick, the state of the game is displayed using simple ASCII graphics.

Step 1: The OOP Payoff: Adding New Features Easily (50 pts)

Object-oriented programming enables the ability to easily add new behavior to the existing objects, incorporate a new class of objects, and connect them into the rest of the program. In this step, you will create a Merchant class whose objects can sail, gather supplies, and be recruited. Then we'll also add a Pirate class whose objects can attack other Sailors, either Merchants or Pirates.

All Sailors will acquire the ability to be "killed;" they will have a "health" value (`double health`), which is decremented whenever they are "plundered." Also, the "health" value decreases when Sailors hide too long in a dock.

When the health hits 0, the sailor will die and go into the "dead" state ('x') from which they cannot emerge. Pirates can be told to attack another Sailor object if the target Sailor object is within range. The Pirate will go into the "attack" state, and on each update cycle, it will hit the target Sailor object, and continue on the next cycle until the target is either in dead or goes out of range.

Class Sailor

In PA3 we saw inheritance (e.g. `Sailor` inheriting from `GameObject`). However in this PA we will take it a step further with objects inheriting from Sailor. You are allowed to add getters or change private variables inside the Sailor class to protected as needed. For this to happen we must "refactor" the Sailor class a little bit:

Protected:

It will be helpful to add a Model reference inside the Sailor class so that in the following parts of PA4 you will be able to add new Sailors to the "world" and you will be able to look for other Sailors at the dock, specifically:

- `Model* world;`

Public members:

- `Sailor` () becomes `Sailor(char);`
  - The default Sailor constructor now must take in a char display code.
  - The Sailor will reference the single model when it is constructed.
  - Leave all other initializations and the output unchanged.
- `Sailor` (int, Dock*) becomes `Sailor(char, int, Dock*);`
  - The Sailor constructor must now take in a char display code along with the other values
  - The Sailor will reference the single model when it is constructed
  - Leave all other initializations and the output unchanged.
- `virtual void start_plunder(Sailor*);`
  - Only print "I cannot plunder."
- `virtual bool start_recruiting(Sailor*);`
  - Only print "I cannot recruit."
- `void get_plundered(int attack_strength);`
  - It subtracts the `attack_strength` from the health, which represents the health of the Sailor. If the resulting new value of energy is less than or equal to zero, print a final message (`"Oh no, now I'm in Davy Jones' Locker!"`), and set the state to 'x'. Otherwise leave the state unchanged and output the message: "Ouch!"
- `virtual double get_speed() = 0;`
  - The `get_speed` function in Sailor must become a pure virtual function, turning the Sailor class into an abstract class.
- `virtual bool update();`

- `case 'h':`
  - The health decreases one quarter. If the health is lower than 5, then the Sailor is officially declared as dead (state `'x'`).
- `case 'x':`
  - Do nothing

Class Merchant

The Merchant "is_a" Sailor, but extends the behaviors of a base class sailor.

Public members:

- `Merchant();`
  - Invokes `Sailor('M');`
  - The initial size of a merchant is 10.
- `Merchant(int id, Dock* home);`
  - Invokes `Sailor('M', id, home);`
  - The initial size of a merchant is 10.
- `double get_speed();`
  - Implements the `Sailor::get_speed()` function
  - Speed of Merchant is **(1/size)*health*4**
- `bool start_recruiting(Merchant *sailor_mate);`
  - Gets called while Merchant is in state 'h'
  - Two merchants can recruit a new merchant if and only if:
    - both Merchants have the same hideout dock, and
    - there are berths available for a new Merchant (even though they are not consumed when hiding), and
    - both Merchants have a health amount between the range of 40 and 60, and
    - there are no other Merchants in the Merchants' home dock.
    - If another Merchant enters the dock while the other Merchants are recruiting, the recruiting process should be stopped.
  - Once that is true the message "I found a new recruit!" is printed by each Merchant
  - Sets the other merchant's state to 'm' so only one new merchant will be created
  - Recruiting takes 2 time ticks and sets the state of both Merchants to 'r'; you can decide how you want to implement keeping track of these 2 time ticks
  - Once mating is over, both Merchant's states are set back to 'h'
- `bool update();`
  - case 'r':
    - checks to see if 2 game ticks have passed and then creates a new Merchant in the same dock
  - All other cases are the same as Sailor. Therefore, reuse the `Sailor::update()` function.
- `void show_status();`

It outputs: “Merchant status: ”, calls Sailor::show_status(), and then displays information about the Merchant specific state of recruiting (see Sample Output).

***** CHECKPOINT I *****

This will involve creating Merchants and showing that they operate just like Sailors, but have a different speed and a different display code. You should also verify the recruiting functionality.

Class Pirate

Pirates can only plunder. They can’t sail. They can’t recruit.

Private members:

- `int attack_strength;`
  - the number of attack points delivered to the target with each "hit." Initial value is 2.
- `double range;`
  - the target must be within this distance to start plundering, and stay within this distance to continue plundering. Initial value is 1.
- `Sailor * target;`
  - the Sailor object being attacked.

Public members:

- `Pirate();`
  - Invokes `Sailor('R');`
  - The initial size of a pirate is 20.
- `Pirate(int);`
  - Invokes Sailor(‘R’, in_id, NULL);
  - The Pirates get generated in a random location. Seed the random number generator using ‘srand’ with the time of the game (the member variable in Model). Then use rand % 20 to generate the X value and the Y value for the location of the Pirate. By seeding the random number generator with the Model game time, the behavior will be consistent between the sample output and yours.
  - The initial size of a pirate is 20.
- `double get_speed();`
  - Implements the `Sailor::get_speed()` function
  - Speed of Pirate is 0 because Pirates do not sail
- `void start_plunder(Sailor* target);`
  - If the distance to the target is less than or equal to the range of 1, output an appropriate message (“Arrrggghhhh!”), save the target pointer, and set the state to 'a' for attack.
  - Otherwise, only output a message: "I will be back for you!".
- `bool update();` This function updates the Pirate object as follows:

- state 'x': do nothing and return false.
- state 's': do nothing and return false;
- state 'a':
  - Check again the distance to the target. If it is out of range, print a message ("Darn! It escaped."), set the target pointer to NULL, set the state to 's', and return true.
  - If it is in range, then check whether the target is still alive. If not, output an appropriate message like "I triumph!", set the state to 's' and return true. If the target is still alive, output a message like "Arrghh matey!" and call the get_plundered function with the attack_strength as an argument, stay in the state, and return false.
  - After the Pirate has sunk a Sailor, it gains energy of 5.

- void show_status();
  - It outputs something like "Pirate status: ", calls Sailor::show_status(), and then outputs whether the object is attacking (see Sample Output).

Class Model

In the constructor of the Model create three Merchant and two Pirate objects instead of the three Sailor objects (from PA3). Put their pointers into the object_ptrs array and the sailor_ptrs array as follows (you will need to update the port pointers to put them in the correctly shifted object_ptrs array):

```
Merchant 1 at Dock 1, object_ptrs[3], sailor_ptrs[0]
Merchant 2 at Dock 2, object_ptrs[4], sailor_ptrs[1]
Merchant 3 at Dock 2, object_ptrs[5], sailor_ptrs[2]
Pirate 1, at location (10, 15), object_ptrs[6],  sailor_ptrs[3]
Pirate 2, at location (15, 10), object_ptrs[7],  sailor_ptrs[4]
num_objects = 12; num_sailors = 5;
```

Also, add a new game command ("plUnder") to the main program:

u ID1 ID2

The sailor with ID1 should start plundering the sailor with ID2. Note that any sailor can be commanded to plunder any other sailor, but only Pirates will actually do it

At this point, you should be able to command Merchants and Pirates to move around, gather supplies, hide, recruit and plunder. When Pirates attack either a Merchant or another Pirate, then their targets should sink (die) if you don't tell them to move away soon enough. You should be able to stage a fight between two Pirates. If you command a Merchant to attack, the Sailor::start_plunder() function should execute, giving you the "We have no cannons!" message. If you tell a Pirate to recruit, the Sailor::start_recruiting() function should execute and output the "My primary goal is to plunder merchants" message.

The Sailor objects all have "zombie" capability. A dead one can still be commanded to do something, but it will not react. If a Sailor is dead, it really shouldn't appear on the display anymore. Define a virtual `is_alive` function in the `GameObject` class that always returns true. In the Sailor class implement the `is_alive` function, which only returns true if the Sailor is alive. Dead Sailors should not occupy space. Then, call the `is_alive` function in the `start_sailing`, `start_supplying`, and `start_plunder` functions to check if the Sailor is dead and print a message to that effect (see Sample-Output).

CHECKPOINT II

Step 2 Refactoring the Model using the STL (30 pts)

In this step, we will be using the Standard Template Library's (STL) "list" container in the Model class to replace the arrays with linked lists. Take a look at your textbook or Internet tutorials (e.g. www.cplusplus.com) to understand how the "list" template from the STL library has to be utilized and make sure that you feel comfortable with using it before proceeding.

First, replace the array of GameObject pointers named "`object_ptrs`" with a linked list named `object_ptrs` and another called `active_ptrs`. The `object_ptrs` list will point to all of the GameObject that exist, while the `active_ptrs` list will point to all of the GameObjects that are still alive and must be updated and displayed. If an object dies, it will be removed only from the active list and will no longer be displayed. Likewise, the `~Model()` destructor must use the `object_ptrs` list to de-allocate all of the objects.

Then, replace the `port_ptrs`, `dock_ptrs`, and `sailor_ptrs` arrays with linked lists as well, and remove the variables like `num_sailors` that were required to use the arrays (the list object comes with functions that tell you the size of the lists). Now, the Model object can handle any number of game objects in any combination of types; there are no longer the artificial size restrictions.

You also need to make the following changes in your Model class:

- In the Model constructor, use the appropriate member functions to put the objects into the list in the same order (front-to-back) as they were in the original array.
- In the `update()` function, you must update each object of the `active_ptrs` list, and then scan the list looking for dead objects by invoking the is_active function of each game object. Dead objects are removed from the `active_ptrs` list so that they will no longer be updated. For debugging and demonstration purposes, output a message like "Dead object removed" (see Sample-Output).
- In the `show_status()` function, iterate over the object_ptrs list to display the status of all the objects, regardless if dead or alive (see Sample-Output).
- The `display()` function should display all objects of the `active_ptrs` list. Therefore, remove any previous code that checked if an object was alive or not since this is now being handled in the `update()` function's modification of the `active_ptrs` list.

Step 3: Use Exceptions to simplify input error handling (20 pts)

Instead of checking for and dealing with invalid user input all over the main program, use exceptions to simplify and centralize the input error handling.

First, define a simple exception class containing a message pointer. Create a file called *InputHandling.h* and put the following class definition in it:

```
class InvalidInput
{
    public:
        InvalidInput (char* in_ptr):  msg_ptr (in_ptr) { }
        const char* const msg_ptr;
    private:
        InvalidInput();    // no default construction
};
```

Since this class is so simple, it does not need any function definitions in a separate source code file.

Second, insert a try block around your code that handles commands, followed by a catch block to handle an InvalidInput exception by printing out the message, taking appropriate action, and then getting the next command. The structure of the code that handles the user inputs would look like:

```
while(command_mode) {
    . . .
    cin >> command;
    try {
        switch(command) {
            . . .
        }
    }
    catch (InvalidInput& except) {
        // actions to be taken if the input is wrong
        cout << "Invalid input - " << except.msg_ptr << endl;
    }
}
```

Third, convert your code to use the exceptions. In the functions of the GameCommand.cpp file add checks for invalid input by throwing an `InvalidInput` exception object containing an appropriate message (see also Sample Output). An example code showing this procedure is given below:

```
void do_anchor_command(Model* m) {
    int id;
    if(!(cin >> id))    // check: is the stream good?
        throw InvalidInput("Was expecting an integer"); // throw an exception
```

```
    Sailor* sailor = m->get_sailor_ptr(id)
    if(sailor == (Sailor*)NULL)  // check: is ID valid?
        throw InvalidInput("Invalid Sailor ID");

    . . .
}
```

CHECKPOINT III

Step 4: Create new objects during program execution (30 pts)

Implement a new game command:

`n TYPE ID X Y`

This command should create a new object with the specified TYPE, ID at location `(X, Y)`.

The two 'n' commands that need to be different are Pirates and Merchants.

For Pirates: **`n TYPE ID`**

For Merchants: **`n TYPE ID ID_HOME`**

TYPE is one character letter abbreviation for the type of object:

- P – Port
- D - Dock
- M - Merchant
- R – Pirate

Specifications:

- In the Model, implement the `add_pointer(GameObject*)` function that adds the GameObject pointer to the appropriate lists depending on the type of the object.
- Implement the `handle_new_command(Model*)` function in `GameCommand.cpp/.h`, which reads the TYPE, ID, and the location. Then, it passes the pointer to the `add_pointer()` function of the Model.
- An unrecognized TYPE code, and invalid inputs for ID, X, and Y should be handled by throwing an `InvalidInput` exception, as described in the previous step.
- Before creating the object, check to make sure that an object with the same ID is not already present; if it is, treat it as invalid input and throw an `InvalidInput` exception. The rules for ID numbers are:
    - Sailor, Ports, and Docks are three separate groups of objects, with their own sets of ID numbers. So as currently the case, you can have a Sailor, a Port, and a Dock all with the same ID number
    - All Sailors should have unique IDs. E.g. Merchant 1, Pirate 2, Merchant 3, Merchant 4, Pirate 5. When a new Sailor is created (when the merchants recruit) it should have the lowest ID available for all Sailors. Example, if the current Sailors are Merchant 1,

Merchant 3 (no merchant 2), the recruited Merchant should be assigned with ID 2.
    - ID numbers may any integer value, but the effects on the grid display when object has an ID number greater than 9 are **undefined (you decide what to do).**
- Lastly, add the new object at the end of the proper lists in the Model. Therefore, implement the `add_new_object(GameObject*)` function in the Model class.

Step 5: Implement persistent objects (30 pts)

A persistent object is an object that persists between runs of the program, or can be removed from memory and then put back in exactly the same state as it was in before it was removed. The standard technique for doing this is to record all of the member variable values for the objects in a file. At some point, the existing objects are destroyed, such as when the program terminates. Then later, a new set of objects is created, and the data in the file is used to restore the member variables to the same values as they used to have. While the new objects are in fact not the "same" as the old objects, they will be in the same state and will be doing the same things as the original objects were doing at the time that the data was saved.

In this project, persistent objects will be used to "save the game" and "restore a game". When the game is saved, the relevant data in the Model object and all of the `GameObjects` will be written to a file. The program can then either continue to run, or be terminated. To restore the game, the file information will be used to recreate a set of `GameObjects` and settings of the Model object that are identical to the situation at the time of the save. Note that restoring a game from a file means that any objects currently existing need to be de-allocated, and the Model needs to be "emptied" of all of the objects.

There is only one complication. It won't work to store the values of pointers in the file and then restore them. Why? Because there is no guarantee that the new operator will place the new objects in exactly the same addresses in memory. In fact, it would be extremely unusual if it did - new finds a convenient piece of memory for you, and exactly where it is depends on a huge number of factors. So for all practical purposes, new gives you an address that you might as well consider to be random.

What to do? The pointers in the various lists and arrays will get set to new addresses anyway - you can restore a list just by building a new one containing the same data items in the same order as the old one. The only pointers that are a problem are member variables in `GameObjects` that are pointers to other `GameObjects`, such as which Merchant a Pirate is attacking, or which Sailor is supplying or anchored. While saving the pointer value is meaningless, all of these objects have id numbers, which should be the same after the game is restored. First save in the file how many objects and the type and id number of each one. Call this information the "Catalog." Then record all of the member variable values for the objects, but if the object contains a pointer to another object, save the id number of the pointed-to object instead of the pointer value.

When it is time to restore the game, first read the Catalog and create those objects with their id numbers. Then restore the member variables of each object using the rest of the data in the file. If the restore code finds an id number for another object, it gets the pointer for that object using the id number. Thus, although the restored objects are all residing in different places in memory, each one ends up with pointers to the correct other objects.

This process is simplified by the standard OOP approach: Make each class responsible for recording and

restoring its own data, taking advantage of the hierarchical structure of the classes, in the same way that previous projects used the `show_status()` function in each class. The `Model` class handles this whole process, because the Model is responsible for managing all of the `GameObject` objects.

Do not save and restore information about dead objects; only objects in the `active_ptrs` list will be saved and restored. Also, do not save and restore the settings of the View class either. Assume that there is no need to detect and handle input or output errors during the save and resume processes. Since the program writes the saved data, it should be able to read the data back in correctly read - no human making typos! You may ignore the remote possibility of hardware malfunctions. If needed, you can assume that no more than 10 objects will be saved or restored.

But there is one critical need! You have to ensure that the data written out of objects and member variables gets read back into the same objects and member variables; since you write and read the data in a stream, the input order of the data has to exactly match the output order.

In this step you have to implement the following two new commands:

> **`S filename`**
> Save the game into the specified filename. You can assume that the filename is a single string of characters (no internal whitespace), and the maximum length of the filename is 99. Ensure to close the file after finishing writing the data to it!
>
> **`R filename`**
> Restore the game using the file specified. If the file does not exist, throw an `InvalidInput` exception. Ensure to close the file after finishing with it!

New member functions:

Provide the following for each class in the `GameObject` hierarchy:

- `virtual void save(ofstream& file);`
  calls the save function for its superclass, then writes to the file the member variables declared in this class. (See PA4's show_status() functions for the same pattern.) If a member variable is a pointer to another GameObject, it writes that objects id number instead. If the pointer is 0, it writes a -1 for the id number (we are now assuming that object id numbers are >=0).
- `virtual void restore(ifstream& fife, Model& model);`
  calls the restore function for its superclass, then reads from the file into the member variables declared in this class. If a member variable was originally a pointer to another GameObject, it reads in that object's id number, and gets the pointer to the new object that has that id number from the model. If the id number is -1, then the value of 0 is stored in the pointer.

Since the restore function contains a reference to the Model class in its prototype, put into each class's header file a "forward declaration" of the Model class:

```
class Model;
```

This is a minimum declaration that is enough to allow you to mention pointers or references to a class, and will help avoid some circular declaration problems. Then `#include "Model.h"` in the `.cpp` file for each class that has to call the Model functions like `get_sailor_ptr`.

Implement the following functions to the Model class:

- `void save(ofstream& file);`

    - Writes the current simulation time into the file.
    - Writes the Catalog information into the file:
        - Outputs the number of objects in the `active_ptrs` list.
        - Goes through the `active_ptrs` list in order from front to end, and outputs a code letter for the type of the object (such as 'R' for Pirate) followed by id number for the object. You can use the object's display_code member variable as the type code if you take into account that it might be either upper or lower case in the object, and it has to be restored to be the same case as it was before. Alternatively, you can use a new member variable to contain a code that is supplied to the object's constructor, and so is completely controlled by the Model.
    - Has each object write its data into the file:
        - Go through the `active_ptrs` list in order from front to end, and call the object's save function.

- `void restore(ifstream& file);`
    - All existing GameObjects are deleted and the corresponding lists and arrays are emptied.
    - Sets the current simulation time using the data in the file.
    - Uses the Catalog data in the file to create a new object of the specified types and id numbers, putting their pointers into the pointer lists and arrays in the original order.
    - Goes through the pointer list in order, and calls the object's restore function.

Further details are up to you. You may find it convenient to define a additional member or non-member functions and/or constructors, another member variable to hold a type code, or another reader or writer function for the classes, and make the corresponding modification where the objects are created. Consider which constructors are actually needed in this project: you can discard or modify the default constructor or other constructors if convenient to clean things up. Be sure that any new or modified constructors will still output the construction message for demonstration purposes. Also, consider overriding the >> operator to make it easier to read in the points and vectors.

Test first just by saving the game immediately after starting it, then looking at the resulting file. Remember that the save file produced here is just a text file, and you can print it, use your programming editor, etc., to inspect the contents of it. **In fact, you will need to submit at least one such save file.** If the contents make sense, try restoring the game from that file. It should be in the same state. Test further, by running the game for a while, making the objects move around and do things, and kill one or two of them. Then save the game, and immediately restore it. You should see the destructor messages for all of the prior objects, and then constructor messages as the new objects are created from the file. Dead objects should not be saved and restored.

CHECKPOINT IV

Step 6: The "Kraken" (40 pts)

As described in the game instructions, the Kraken appears every 15 time ticks and sweeps away all Sailor objects – either dead or alive – that are not in a dock.

It is your task to implement the Kraken without any restrictions. However, you must comply with the following requirements:

- Kraken should be an independent class. That is, don't mix the Kraken's behavior into any class. .
- 5 time ticks before the Kraken arrives, you have to print a message "Early Kraken Warning!"
- After the Kraken, print out all Sailors that have been swept away (see Sample Output).
- At the end of the game, print out all Sailors that the Kraken has swept away (see Sample Output)
- There can be only one Kraken object. Hint: "Singleton" Pattern
- You're not allowed to cheat! This is actually true for the whole Programming Assignment. ☺

How would you implement such behavior?

Submission

Please use the file names **CartPoint.h, CartPoint.cpp, CartVector.h, CartVector.cpp, GameObject.h, GameObject.cpp, Port.h, Port.cpp, Dock.h, Dock.cpp, Sailor.h, Sailor.cpp, Merchant.h, Merchant.cpp, Pirate.h, Pirate.cpp, Kraken.h, Kraken.cpp, View.h, View.cpp, Model.h, Model.cpp, GameCommands.h, GameCommands.cpp, InputHandling.h, PA4.cpp, mySaveFile.txt, and README.txt**. Put all your `.cpp` files in a folder named <your username>_PA4 (e.g., dougd_PA4), zip it, and submit a single file (e.g., dougd_PA4.zip) following the submission guidelines on Blackboard. Do **NOT** submit your executable files (a.out or others) or any other files in the folder. Make sure to **comment** your code.