Programming Assignment 4 (PA4)
Dynamic Allocation, Program Organization, Exceptions, Save/Load, and STL

Out: November 10, 2019, Monday -- DUE: December 11, 2019, Monday, 11:59pm
EC327 Introduction to Software Engineering – Fall 2019

Total: 100 points + 80 extra credit

- *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.*
- ***No late submissions for PA4 will be accepted**.*
- *Follow the assignment submission guidelines in this document or you will lose points.*

Submission Format (please read)

- Use the exact file names specified in each section for your solutions.
- Complete submissions should have **~27 files (depending on extra credit done or not).** 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). INCLUDE YOUR MAKEFILE.
- 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!)

Overview

This assignment is presented as a series of steps to build upon PA3 by creating new subclasses of Game_Object, adding the ability to spawn new game objects on the fly (at runtime) and changing the underlying data structures in the model view controller (MVC) framework. We recommend you follow these steps to arrive at the final result in the easiest, most logical, and incremental manner. You should have a running, working program at the end of each step, so you can check your progress, debug your code, and have some fun playing the game at each point. Despite these steps, ***only the final version should be turned in for full credit.***

Note: These specifications are very specific when they need to be, and looser where the design is up to you. You should read this document carefully (and completely) in order to get full credit for your PA4. The basic rule: if not specified, you can do the required coding anyway you want; but please ask in case of doubt regarding program function. Be sure to check announcements on blackboard and discussions in Piazza often for any questions and clarifications.

Step 1: Adding New Features Easily (40pts)

Thanks to inheritance, polymorphism, and the MVC pattern, at this point it should be easy to add new behaviors 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 an opponent class for the training units inside the Battle Arenas. When you beat all the arenas, you will be a champion and to do that you need to beat all of your rivals.

All Pokemon will acquire the ability to be "fainted". They will have a "health" value which is decremented whenever they are "hit." When the health hits 0, the object will "die" and go into state ‘FAINTED’ from which they cannot emerge. Now that Pokemon can die, alter your code so that when they run out of health, the state is set to ‘FAINTED’ instead of ‘I’.

Class Pokemon

Modify the Pokemon class as follows:

- *Add New State*
  - IN_ARENA = 9
  - MOVING_TO_ARENA = 10
  - BATTLE = 11
  - FAINTED = 12
- *New Protected members*
  - double health
    - Initial value is 20
  - double store_health
    - It will be same as health and will be used to restore health if pokemon wins the battle.
  - double physical_damage
    - Initial value is 5
  - double magical_damage
    - Initial value is 4
  - double defense
    - It will parry percentage of the attack that Pokemon received.
    - Default value is 15
  - Rival* target
    - The rival that we will battle in the arena
  - Bool is_in_arena

        - Returns true if the Pokemon inside the arena
    - BattleArena * current_arena
        - Holds the current arena

- *Public members:*
    - Pokemon(string in_name, double speed, double hp, double phys_dmg, double magic_dmg, double def, int in_id, char in_code, Point2D in_loc): GameObject('P',id, in_loc)
        - Update the constructor with the new protected members
        - Also update the Pokemon declarations in the model.cpp
    - bool IsAlive()
        - It returns true if the state is not 'FAINTED', false if it is 'FAINTED'

    - void TakeHit(double physical_damage, double magical_damage, double defense)
        - This function will be called from StartBattle() function until either Pokemon or Rival has 0 or less than 0 health
        - It subtracts the chosen attack type's damage from the health.
        - You should choose the attack type with a random function. *Hint: Use rand()*
        - Damage is calculated with the following rules
            - Defense is percentage of the damage that you will parry
            - Basically effective damage is:

```
damage = (100.0 - defense)/100 * physical_damage;
```

    - void ReadyBattle(Rival *in_target)
        - If the state equals "IN_ARENA", current_arena->IsAbleToFight() is True, current_arena->IsBeaten() is False and in_target->is_alive() True
            - Set the target = in_target
            - State is "BATTLE"
        - Else
            - Set the state "IN_ARENA"
    - bool StartBattle()
        - Call this function in the BATTLE state
        - Assume that we are consuming one time step during the battle loop
        - This function holds the battle strategy, it calls take TakeHit() functions for the Pokemon and the rival in a loop until Pokemon or Rival ends up with health less or equal than zero
        - Always rival hits first
    - Pokemon::Update()
        - If the state is 'FAINTED', stay in that state, and do nothing except return false.

- Add case statement for MOVING_TO_ARENA , BATTLE and FAINTED states
    - MOVING_TO_ARENA
        - Same as MOVE_TO_GYM or CENTER
    - BATTLE
        - This will be the step after ReadyBattle() function is called.
        - Decrease stamina and Pokemon dollars according to the arena costs.
        - Call StartBattle() function
            - If the pokemon wins:
                - Refill the pokemon health
                - State = IN_ARENA
                - Call target->IsAlive() function to update target's state
            - Else:
                - State= FAINTED
                - Call target->IsAlive() function to update target's state

Add a new command to the main program:

b ID1 ID2 - battle - command Pokemon ID1 to start battle with Rival ID2. Check for invalid id numbers and input errors as with the other command functions. Note that only Pokemons can start the battle inside the arena.

a ID1 ID2 - move to arena command Pokemon ID1 to move the arena with ID2. Check for invalid id numbers and input errors as with the other command functions. Very similar command that you implemented for gym.

Class Rival

Derive a new class from the GameObject: Rival

- *New Protected members*
    - double health
        - Initial value is 20
    - double physical_damage
        - Initial value is 5
    - double magical_damage
        - Initial value is 4

    - double defense
        - It will parry percentage of the attack that Rival received.
        - Default value is 15
    - Bool is_in_arena
        - Returns true if the rival inside the arena
    - BattleArena * current_arena
        - Holds the current arena
- Public members:
    - Rival::Rival(string name, double speed, double hp, double phys_dmg,double magic_dmg, double def, int id, Point2D in_loc) :GameObject('R', id, in_loc))
    - void TakeHit(int physical_damage, int magical_damage, int defense)
        - Same as the take_hit function that Pokemon class has
    - double get_phys_dmg()
        - Return physical damage
    - double get_magic_dmg()
        - Return magical damage
    - double get_defense()
        - Return defense
    - double get_health()
        - Return health
    - bool Update(); This function updates the Rival object as follows:
        - state ‘ALIVE_RIVAL’: This is the default state when we create rivals and it returns false;
        - state ‘FAINTED_RIVAL’: When the rival’s health is 0 or below
    - void ShowStatus();
        - It outputs something like "Rival status:", then functions similar to Pokemon::show_status() where it calls Game_Object::show_status and that outputs state specific information, for example if the Rival is alive or not alive.
    - bool IsAlive()
        - Check the health of rival and return True if it is alive

Modify the Model class to create two rival objects and put their pointers in the object_ptrs array and an rival_ptrs array as follows:

Rival 1, (x, y), object_ptrs[6], rival _ptrs[0]

Rival 2, (x1, y1), object_ptrs[7], rival _ptrs[1]

num_objects =8; num_rivals = 2;

We need to create them inside the battle arena, so their location should be same as Battle Arena’s location.

Creating the Battle Arena (10 pts)

This class has a location and a set amount of rivals. It also has a display_code letter and an id number that are used to help identify the object in the output. BattleArena inherits from Building.

```
enum BattleArenaStates {
        RIVALS_AVAILABLE       = 0,
        NO_RIVALS_AVAILABLE = 1
};
```

Private Members

- unsigned int max_num_rivals
    - The maximum number of rivals this Battle Arena can hold.
    - Initial value should be set to 3.
- unsigned int num_rivals_remaining
    - The number of rivals currently in this Battle Arena
    - Initial value should be set to num_rivals
- double dollar_cost_per_fight
    - The per fight cost in Pokemon Dollars
- unsigned int stamina_cost_per_fight
    - Stamina cost for single battle in the arena
- unsigned int pokemon_count
    - Number of pokemons inside the arena

Constructors

- The default constructor that initializes the member variables to their initial values:
    - display_code should be ‘A’
    - pokemon_count should be 0
    - num_rivals should be 3
    - num_rivals_remaining should be set to num_rivals

    - dollar_cost_per_fight should be set to 4
    - Stamina_cost_per_fight should be set to 3
    - state should be RIVALS_AVAILABLE.
    - Should print out the message “BattleArena default constructed”.

- BattleArena(unsigned int max_rivals, unsigned int stamina_cost, double dollar_cost, int in_Id, Point2D in_loc): Building('A', in_id, point2d)
    - Initializes the id number to in_id, and the location to in_loc, and remainder of the member variables to their default initial values.
    - Prints out the message “BattleArena constructed”.
    - state should be RIVALS_AVAILABLE.

Public Member Functions

- unsigned int GetNumRivalsRemaining()
    - Returns the number of rivals remaining in this Battle Arena.
- bool HasEnoughRivals()
    - Returns true if this Battle Arena contains at least one rival.
    - Returns false otherwise.
- double GetDollarCost()
    - Returns the cost of one battle in the arena.
- unsigned int GetStaminaCost()
    - Returns the amount of stamina required for battle
- bool IsAbleToFight(double budget, unsigned int stamina)
    - Returns true if a Pokemon with a given budget and stamina can request to fight
    - Returns false otherwise
- bool Update()
    - Returns false if rivals still remain within the BattleArena.
    - This function shouldn’t keep returning true if the BattleArena has no rivals left. It should return true ONLY at the time when the BattleArena has 0 rivals left, and return false for later “Update()” function calls.
- bool IsBeaten()
    - Returns true if num_rivals_remaining == 0
- void ShowStatus()
    - Prints out the status of the object by calling GameObject’s show status and then the values of its member variables:
        - Calls Building::ShowStatus()
        - “Max number of rivals: <num_rivals>
        - Stamina cost per fight: <stamina_cost_per_training>
        - Pokemon dollar per fight: <dollar_cost_per_fight>
        - <num_rivals_remaining> rival(s) are remaining for this arena

- See the sample output for the format.

When you are creating a Rival object, make sure that you point the Battle Arena by setting BattleArena* curren_member of rival in the constructor. Whenever a Rival is beaten, you can write a similar function to RemoveOnePokemon() to update remaining rival count. You can also come up with a different solution.

Step 2: Replace your arrays with Linked Lists using the STL Library (20pts)

In this step, you will be using the Standard Template Library's (STL) "list" container to replace your arrays. Take a look at your textbook or internet tutorials to understand how the "list" template from STL library is used and make sure that you feel comfortable with using it before proceeding. **Don't worry. It is pretty easy to understand.**

First, replace your array of Game_object 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 Game_Objects that exist, while the active_ptrs list will point to all of the Game_Objects that are still alive and must be updated and displayed. If an object dies, it will be removed from the active list and will no longer be displayed. However, if we list the status of all the objects, all of them will be listed. Likewise, of course, ~Model() will use the object_ptrs list to deallocate all of the objects.

Then replace the pokemon_ptrs, rival_ptrs, gym_ptrs, arena_ptrs and center_ptrs arrays with linked lists as well, and remove the variables like num_rivals 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 restrictions of array sizes.

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.
- Model::update() should update each object in the active_ptrs list, and then scan the list looking for dead objects; if found, the dead object is removed from the active_ptrs list so that it is no longer updated. For debugging and demonstration purposes, output a

message like “Dead object removed”.

- Model::display() should display the objects in the active_ptrs list, so remove any previous code that checked for whether an object was alive or not – this is now handled in the update()
- Model::show_status() should display the status of all of the objects in the object_ptrs list.

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

In this step, 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 *Input_Handling.h* and put the following class definition in it:

```
class Invalid_Input
{
    public :
        Invalid_Input(string in_ptr) : msg_ptr (in_ptr) { }
        const string  msg_ptr;
    private :
        Invalid_Input ();
    // 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 Invalid_Input 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 (Invalid_Input& except){
        cout << "Invalid input - " << except.msg_ptr << endl;
        // actions to be taken if the input is wrong
}
```

Third, convert your code to use the exceptions. Everywhere in your code that you check for invalid input, instead of doing whatever on-the-spot cleanup and error handling, create and throw an Invalid_Input exception object containing an appropriate message. An example code showing this procedure is given below:

```
int get_int(){
        int i;
        if(!(cin >> i))       // do the input, then check: is stream good?
                Throw Invalid_Input("Was expecting an integer"); // throw an exception
        return i;
}
```

The error-handling has now been removed from the normal flow of control; it no longer clutters the scene!

NOTE: You can modify these exceptions if needed (using strings, inheritance, etc.). The point is that you need to implement exception handling for user input. Feel free to be creative if you want.

Step 4: Create new objects during program execution (20pts)

Implement a new command:

n TYPE ID X Y- create a new object with the specified TYPE, ID number, (X, Y) location

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

- g – PokemonGym
- c – PokemonCenter
- p – Pokemon
- r – Rival

Specifications:

- Implement the command by calling a new function defined in Model, NewCommand(). This function reads the TYPE, ID, and other information, creates the new object and adds its pointer to the appropriate lists depending on the type of the object.
- Add new objects at the end of the lists.
- An unrecognized TYPE code, and invalid inputs for ID, X, and Y should be handled by throwing an Invalid_Input exception, as in Step 4.
- Before creating the object, check to make sure that an object with the same ID number is not already present; if it is, treat it as invalid input and throw an exception. The rules for ID numbers:
  - Pokemons, Rivals, Pokemon Centers, and Pokemon Gym are four separate groups of objects, with their own sets of ID numbers. So as currently the case, you can have a Pokemon, a Rivals, a Pokemon Center and a Pokemon Gym all with the same ID number.
  - Within each group of objects, ID numbers must not be duplicated. So for example, you may not have a two Pokemon Gyms with an ID of 1.
  - ID number may have 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).**
- If everything is valid, you can call the default constructor or any other constructor that you prefer.

EXTRA CREDIT – The Red Steps Are Extra Credit

Step 5: Implement persistent objects (30pts)

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 are 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 Game_Objects 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 Game_Objects 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 deallocated, 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 Game_Objects that are pointers to other Game_Objects, such as which Pokemon a Rival is attacking, or which Pokemon_Center a Pokemon recovering its' stamina. 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. Finally, this whole process is handled by the Model class, because it is responsible for managing all of the Game_Object objects.

For simplicity, do not save and restore information about dead objects; only objects in the

active_ptrs list will be saved and restored.

For simplicity, **do not** save and restore the settings of the View class either. Also for simplicity, assume that there is no need to detect and handle input or output errors during the save and restore processes. Note that since the program writes the save data, it should be able to depend on it being correctly read back in - 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 be sure 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.

The requirements for this step follow:

New commands:-

- S filename- Save the game in the file specified. You can assume that the filename is a single string of characters (no internal whitespace), and its maximum length is 99 characters. Be sure 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 Invalid_Input exception. Be sure to close the file after finishing with it.

New member functions:

Provide the following for each class in the Game_Object 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 Game_Object, 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& file, 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 Game_Object, 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_Pokemon_ptr.

Add the following functions to the Model class:

- void save(ofstream& file); does the following:
    - 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 'A' for Battle Arena) 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); does the following:
    - All existing Game_Objects 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.

The details are up to you. You may find it convenient to define an additional or different constructor, 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 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 restored.

Step 6: Finish with some fun (50pts)

Let's make the objects behave a little more interestingly, with only a few lines of code. Implement the following:

- (10) Improve battle function: You should add different attack skills to your Pokemon such as "ThunderShock", "ThunderStorm", "Quick Attack", etc.
    - You should have a list of attacks and when you generate a Pokemon or Rival, they need to inherit these attacks.
    - You should have separate attack list for each Pokemon and rival.
    - Each attack needs to decrease specific amount of HP and different magical/physical attack damage.
- (10) Add new functionality to gym:
    - After you gain enough experience points, develop a mechanism that can be used to level up your physical and magical damage.
        - For example, if you spend 7 experience points you can increase your physical damage by 1.
    - This should be available in the Pokemon Gym.
    - Pokemon stats need to be updated.
- (20) Computer Mode: Set up the game so that the computer can control the activity of the Rivals and the user controls the activity of the Pokemons.
    - At the start of the game the user should be asked if they wanted to enter Normal mode or computer mode.
    - In computer mode, the user can enter a command and then enter go or run. After this, the computer should execute its own actions.
    - Computer can play with rivals and train them in the Pokemon Gym, when there is another Pokemon exists in the Battle Arena, it can fight with the Pokemon.
- (10) Can you come up with any other cool features? Implement something new and explain it.

If you choose to do any of these extra credit problems, please make a video that is less than 3 minutes that shows how to play the game and describes the code you implemented. Post this video to the class YouTube page. Also, add a comment to the top of the header file describing which steps you did. This includes Step 6 and 7.