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

Out: November 22, 2021, Monday -- DUE: December 8, 2021, Wednesday, 11:59pm
EC327 Introduction to Software Engineering – Fall 2021

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)**. Submit all your files to the PA4 GitHub. A repository and associated links will be created and posted to Piazza. 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 A New Class (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 “Virus”. Viruses are on the game board and if a Student comes “into contact” with them (defined by you), they lose antibodies faster (how much faster is up to you).

Class Virus

Derive a new class from the GameObject: Virus

- *New Protected members*
  - double virulence
    - Initial value is 5
  - double resistance
    - Initial value is 2
  - bool variant
    - Initial value is false
    - For extra credit you could actually make a derived class of Virus
  - double energy
    - Initial value is 10
  - bool is_in_student
    - Returns true if the virus is inside a Student
    - Default is false
  - string name
  - Student * current_student
    - Holds the current student it is infecting

- *Public members:*
  - Virus::Virus(string name, double virulence, double resistance, double energy, bool variant, int id, Point2D in_loc):GameObject('V', id, in_loc))
  - void infect(Student* s)
    - This infects a Student
    - This should both change the Virus’ current_student value and update the student’s information as needed as well
  - bool get_variant()
    - Return variant
  - double get_virulence()
    - Return virulence
  - double get_resistance()
    - Return resistance
  - double get_energy()
    - Return energy

- bool get_in_student()
    - Returns is_in_student;
- bool Update(); This function updates the Virus object as follows:
    - state 'IN_ENVIRONMENT: This is the default state when we create a Virus and it returns false;
    - state 'DEAD: When the Virus' energy is 0 or below
    - state 'IN_STUDENT': when the Virus is in a student
- void ShowStatus();
    - It outputs something like "Virus status:", then functions similar to Student::show_status() where it calls Game_Object::show_status and that outputs state specific information, for example if the Virus is alive or not alive. **Staff can provide specific details via demo video or on request.**
- bool IsAlive()
    - Check the energy of a virus and return True if it is alive

Modify the Model class to create two virus objects and put their pointers in the object_ptrs array and the virus_ptrs array as follows:

Virus1, (10, 12), object_ptrs[6], virus _ptrs[0]

Virus2, (15, 5), object_ptrs[7], virus _ptrs[1]

num_objects =8; num_virus = 2;

Now that you have made a virus you have three different choices for how you can modify your program (in order of difficulty)

1. Modify your program so that if a virus and student are on the same location, the virus infects the student.
    a. This requires that you set the Student in the Virus object
    b. You modify student as needed to say they have a Virus – you will likely have to add material to the Student class
    c. When the Student moves it also moves the Virus with it – when a Student moves check if they have a Virus and if so, move it as well.
    d. When the student does anything its antibodies are affected differently if they have a Virus
        i. You can do this however you want based on the variety of parameters in the Virus. **Be creative!**
2. Modify your program so that if a Student comes into some proximity with a Virus, the Student is infected.
3. Modify your program so that Viruses can move randomly (and then implement #1 or #2)
4. Make your Viruses smart so they seek out Students (and then implement #1 or #2)
5. Other??

Describe in your README file what you decided to do.

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 student_ptrs, virus_ptrs, class_ptrs, and office_ptrs arrays with linked lists as well, and remove the variables like num_students 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:

- d – DoctorsOffice
- c – ClassRoom
- s – Student
- v – Virus

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:
    - Students, Virus, DoctorsOffice, and ClassRooms are four separate groups of

objects, with their own sets of ID numbers. So as currently the case, you can have a Student, a Virus, a DoctorsOffice and a ClassRoom 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 DoctorsOffices 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 Student a Virus is infecting, or which DoctorsOffice a Student recovering its' antibodies in. 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:

<u>New commands:-</u>

- <u>S filename</u>- 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.
- <u>R filename</u> - 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_Student_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 you are doing this problem.** 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) Add another class: You can think about what other classes you want to create. These could include "GradStudents" that are more immune (or less immune) to getting infected, "Hospitals" that are like DoctorsOffices but have more features, or derived classes of Virus that are more powerful, can move, seek out students, etc. You could also add classes that move around and kill Viruses, provide ClassRooms with more assignments, or more vaccines to DoctorsOffices. Be creative.

- (20) Computer Mode: Set up the game so that the computer can control the activity of the Virus and the user controls the activity of the Students.
    - 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 Virus. This can include making new ones and having the do actions.

- (10) Add new functionality to the other classes:
    - Have ClassRooms get more assignments overtime (replenish assignments).
    - Have DoctorsOffices have "hours". These would be some multiple of the "ticks" in the game. They can then be open or closed.
    - Have Students become "Immune" to Viruses after some criteria are met.

    - Have Viruses multiply and create new Viruses after time.

- (10) Can you come up with any other cool features? Implement something new and explain it in your Readme.

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.