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

Out: November 28, 2016, Monday -- DUE: December 12, 2016, Monday, 11:59pm
EC327 Introduction to Software Engineering – Fall 2016

Total: 100 points + 60 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 **~29 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).
- 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 (30pts)

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 a Soldier class who can move around and attack other Persons, either Miners or Soldiers. All Persons will acquire the ability to be "killed". 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 'x' from which they cannot emerge. Soldiers can be told to attack another Person object; if the target object is within range, the Soldier will go into the attack state, and then on each update cycle, it will hit the target object, and continue on the next cycle until the target is either dead or goes out of range.

Add the following to the Person class:

Class Person

Private members:

- int health
    - initial value is 5

Public members:

- bool is_alive()
    - It returns true if the state is not 'x', false if it is 'x'
- void take_hit(int attack_strength)
    - It subtracts the attack_strength from the health. If the resulting new value of health is less than or equal to zero, print a dramatic final message ("Arrggh!"), and set the state to 'x'. Otherwise, leave the state unchanged and output an appropriate message: "Ouch!"
- modify show_status to show health if alive, "is dead" if not.
- virtual void start_attack(Person * target)

- It does nothing but output a message: "I can‘t attack."

Modify the Miner class as follows:

- Miner::update();
  - If the state is ‘x’, stay in that state, and do nothing except return false.

Write the Soldier class with its own .h and .cpp files to inherit from Person with the following members:

Class Soldier

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 attacking, and stay within this distance to continue attacking. Initial value is 2.0.
- Person * target;
  - the object being attacked.

Public members:

- void start_attack(Person * in_target);
  - if the distance to the target is less than or equal to the range, output an appropriate message (“Clang!”), save the target pointer, and set the state to 'a' for attack. If it is too far away, output a message: "Target is out of range" and do nothing.
- bool update(); This function updates the Soldier object as follows:
  - state ‘x’: do nothing and return false.
  - state ‘s’: do nothing and return false;
  - state ‘m’: same thing as in the Miner::update() function for this state.
  - state ‘a’: Check the distance to the target. If it is out of range, print a message, 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 “Clang!" and call the target's take_hit function with the attack_strength as an argument, stay in the state, and return false.
- void show_status();
  - It outputs something like "Soldier status:", then calls Person::show_status(), then outputs whether the object is attacking.

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

Soldier 3, (5, 15), object_ptrs[5], person _ptrs[2]

Soldier 4, (10, 15), object_ptrs[6], person _ptrs[3]

num_objects =7; num_persons 4;

Add a new command to the main program:

a ID1 ID2 - attack - command Person ID1 to start attacking Person ID2. Check for invalid id numbers and input errors as with the other command functions. Note that any Person can be commanded to attack any other Person, but only Soldiers will actually do it (hence why you need the virtual function start_attack in Person).

At this point, you should be able to command both Miners and Soldiers to move around, Miners to work, and Soldiers to attack. Soldiers should be able to attack both Miners and other Soldiers, and their targets should die if you don‘t tell them to move away soon enough. You should be able to stage a fight between two Soldiers. If you command a Miner to attack, the Person::start_attack function should execute, giving you the "I can't attack" message. If you tell a Soldier to work, the Person::start_mining function should execute, giving you the "I can't work" message (make sure you have a virtual start_mining function in Person as well for this purpose).

Step 2: Cleaning Up (10pts)

The Person objects all have "zombie" capability. A dead one can still be commanded to do something, and it will. Insert a check in the start_moving, start_mining, and start_attacking functions so that if the object is dead (state ‘x’), it prints a message to that effect, and ignores the command. The messages are specified in the sample output. Check out the sample output to make sure you have done it correctly.

If an object is dead, it really shouldn't appear on the display anymore. Define the is_alive function to be virtual in the Game_Object class and to always return true. Then fix model::display() so that a dead object is not plotted on the grid.

Check your program out to see if you still have zombies and corpses littering the scene.

Additional Specifications

All I/O must be done with C++ iostream facilities. No C-style I/O is permitted (you likely are not using it anyway). All input must be done with the stream input operator >> except that the get function can be used to skip to the end of a line following an input error.

When your program starts, it must output our standard header information that identifies the output, and must write some sort of terminating message when it stops. The constructor messages may appear either before or after the header information, and the destructor messages can appear after the final terminating message.

The messages are specified in the sample output. Check out the sample output to make sure you have done it correctly.

Step 3: 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 hall_ptrs, mine_ptrs, person_ptrs arrays with linked lists as well, and remove the variables like num_persons 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() function's modification of the active list.
- Model::show_status() should display the status of all of the objects in the object_ptrs list.

Step 4: 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 (char  *  in_ptr)   :  msg_ptr (in_ptr) { }
         const  char  *  const 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 5: 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 – goldmine
- t – townhall
- m – miner
- s – soldier

Specifications:

- Implement the command by calling a new function defined in Model, handle_new_command(). 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:
  - Persons, Gold_Mines, and Town_Halls are three separate groups of objects, with their own sets of ID numbers. So as currently the case, you can have a Miner, a Gold_Mine, and Town_Hall all with the same ID number.
  - Within each group of objects, ID numbers must not be duplicated. So you may not have a two Gold_Mines with an ID of 1, or a Miner with an ID of 3 and a Soldier with an ID of 3.
  - 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).**

EXTRA CREDIT – The Red Steps Are Extra Credit

Step 6: Add a new kind of object – Inspector (30pts)

One of the advantages of Object-Oriented Programming (OOP) compared to conventional programming styles is that it is relatively easy to add new abilities or features to a program. In this last step, you will practice doing this by adding a new kind of Game_Object. While its behavior is fairly complex, you should be able to implement it relatively easily by taking advantage of the facilities already present in your program, including lists. However, design work is required for the implementation, so be sure to allow time for it.

An Inspector inspects the Gold Mines.

You will implement a new kind of Person object called an "Inspector." The inspector inspects mines, visiting each one and reporting how much gold is present in each mine when it arrives. After visiting all of the mines, it retraces its path, visiting all of the mines in reverse order, and reports the difference in the amount of gold present on its outbound visit and on its inbound visit. Naturally, this will be zero unless

some Miners have been at work.

Inspector behavior:

- Outbound phase:
  When commanded to start inspecting, an Inspector object begins its outbound phase by moving to the closest Gold_Mine. On the next update after its arrival, it reports the amount of gold currently in the mine and tells itself to begin to move to the next mine. This next mine is the one that is closest to its current position that it has not already visited. It repeats the process until it has reported on the last mine.

- Inbound phase:
  After reporting on the last mine, the Inspector then begins the inbound phase of its inspection trip. It returns to the previous mine it had visited, and accesses the amount of gold in it. The Inspector reports on the difference in the amount of gold between its outbound and inbound visit. For example, if the mine had 70 of gold when the Inspector visited it outbound, and 30 when it visited it again inbound, then it would report that 40 gold has been removed from the mine. It continues to visit each mine, in the reverse order to its original visit, and reports on the difference in the amount of gold between its outbound visit and its inbound visit finally returning to its original starting point and stopping.

Detailed specifications:

- Only mines in existence at the time of the start_inspecting command should be inspected.
- The Inspector::update function should return true only when the Inspector arrives at a destination.
- If the game is saved (Step 7), and then later restored the Inspector must resume and continue any inspection trip that was underway at the time the game was saved.
- If the inspector gets a command during its inspection trip, it stops its inspection trip and goes on to perform the new command. If it is then told to start inspecting again, all memory of its previous trip is erased, and it starts an entirely new trip from its current location. In other words, the command to start inspecting again erases any records from previous, interrupted inspection trips. **Note:** Accordingly, if an inspection trip is interrupted with a command to start an inspection trip, the previous trip is forgotten, and a new trip is started from the Inspector's current location.
- The Inspector should print a message and stop if told to inspect in a situation where an inspection is not possible, such as no Gold_Mines exist.
- The new commands:
  - n i ID X Y - create a new Inspector.
    i ID - command Person ID to start inspecting.

- Add a new object to Model() that creates an Inspector with ID = 5, at location (5, 5) and added to the end of all of the appropriate lists.
- Messages printed should be something like the following:
  - "Starting inspection trip" in response to the command.
  - "This mine contains 70 of gold" for each outbound report.
  - "Starting back" after reporting on the last mine.
  - "Amount of gold removed from this mine is 70" for each inbound report.
    - Notice that the last mine visited on the outbound part of the trip does not get a "removed" report.
  - "Inspection trip completed." after returning to the original location.

- show_status messages should be similar to that of Miner, consisting of moving inbound/outbound, and reporting inbound/outbound.

Implementation requirements:

- The Inspector class must inherit from Person.
- New public functions:
    - Define a reader function for Gold_Mine, double get_gold_amount(), that returns the amount of gold currently in the mine.
    - Define virtual void Person::start_inspecting(Model&), which outputs a message like "Sorry, I can't inspect", and override it in the new Inspector class.
    - Define a reader function in Model to return a copy of the mine pointers list:

```
std::list<Gold_Mine *> get_Gold_Mine_list() ;
```

The rest of the Inspector implementation is up to you, but you may not modify any of the other classes beyond the specification and requirements listed above.

Hints:

Start by considering why the Model is a parameter of the start_inspecting function. What does the Inspector need to know in order to do its work?

Consider using three list variables in Inspector: one for the mines not yet inspected, the other used as a Last-in/First-out (LIFO) stack for the mines that have been inspected on the outbound portion of the trip, and a corresponding list to stack the amounts that were in each mine on the outbound trip.

Roughly speaking, the functions for this class will be similar to the Miner class. Use inherited and virtual functions in the same way as used in Miner and Soldier; only write code for behavior that is specific to Inspectors.

Step 7: Implement persistent objects (20pts)

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 Person a Soldier is attacking, or which Gold_Mine a Miner is working. 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_Person_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 ‘G’ for Gold_Mine) 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 8: Finish with some fun (10pts)

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

- If a Person's health declines below 3, its display code will change to lower case.

- If a Miner or other Person gets attacked, it will run away. If a Soldier gets attacked, it will start attacking its attacker. How? Modify the take_hit function in Person to be virtual and with a second argument which is the attacker's "this" pointer:

  ```
  virtual void take_hit(int attack_strength, Person *attacker_ptr);
  ```

  - In Person, this function calculates a vector between this object's location and the attacker's location, multiplies by 1.5, and then tells itself to move there. Get the sign right, and the Miner runs away from the attacker, not towards it. Now it's hard to kill those guys - have to chase 'em all over the place!

  - Soldier overrides this function to tell itself to attack the attacking person. Mayhem results!

  - In both cases, remember to ensure that the object's health gets updated first; the object shouldn't respond to the attack if it has just died.