Programming Assignment 3 (PA3)
The Gold Standard: Mining in C++ – Episode 1
User-Defined Types, Encapsulation, Operator Overloading, Interacting Objects
Out: October 3rd, 2016, Wednesday-- DUE: November 28, 2016, Monday, 11:59pm
EC327 Introduction to Software Engineering – Fall 2016

Total: 200 points

- *You may use any development environment you wish, as long as it is ANSI C++ 11 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.*
- *Labs may be submitted up to a week late at the cost of a **30% fixed penalty** (e.g., submitting a day late and a week late is equivalent). It is in your best interest to complete as many lab questions as possible before the deadline. If you have missing questions in your original submission, you may complete and submit the missing solutions during the following week. Any submissions after the deadline will be subject to the 30% penalty. No credit will be given to solutions submitted after the 1-week late submission period following the deadline.*
- *Follow the assignment submission guidelines in this document or you will lose points.*

Submission Format (Must Read)

- Use the **exact** file names specified in each problem for your solutions.
- Complete submissions should have **20 files.** Put all of your files in a single folder named: **<your username>_PA3** (e.g., dougd_PA3), zip it, and submit it as a single file (e.g., dougd_PA3.zip).
- Submit your .zip to the PA3 portal on Blackboard by 11:59pm on the due date.
- 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. Otherwise, this is an automatic zero.**
- Comment your code (good practice!). We **may** use your comments when grading.

Coding Style (reminder from PA2)

As you become an experienced programmer, you will start to realize the importance of good programming style. There are many coding "guidelines" out there and it is important that you become to adopt your own. Naming conventions will help you recognize variable names, functions, constants, classes etc. Similarly, there are many ways to elegantly format your code so that it is easy to read. All of these issues do not affect compilation and hence are easy to overlook at this stage in your development as a programmer. However, you ability (or inability) to create clean, readable code could be the difference in your career when you leave college.

Good reference: http://google-styleguide.googlecode.com/svn/trunk/cppguide.html
If you find any other good references, feel free to post the link on Piazza!

The Gold Standard: Mining in C++ Overview

In this programming assignment, you will be implementing a simulation similar to "Warcraft," consisting of objects located in a two-dimensional world that move around and behave in various ways. The user enters commands to tell the objects what to do, and they behave in simulated time. Simulated time advances one "tick" or unit at a time. Time is "frozen" while the user enters commands. When the user commands the program to "go", time will advance one tick of time. When the user commands the program to "run", time will advance several units of time until some significant event happens (to be described later).

In this assignment, you will be implementing these 8 classes:
- Cart_Point – represents a point on a Cartesian coordinate system.
- Cart_Vector – represents a vector in the real plane.
- Game_Object – base class for all objects in the game.
- Gold_Mines – gold mines contain a certain amount of gold. Inherits from Game_Object
- Town_Halls – locations that can store gold.
- Person – inherits from GameObject.
- Miner – a simulated person that can do two things:
  - Move to a specified location and stop.
  - Move gold from a specified Gold_Mine to a specified Town_Hall, shuttling back and forth until the mine is empty.
- View - Displays game objects. More details to come (*to be described later*).
- Model - Holds references to game objects. Model details to come (*to be described later*).

You will also provide a set of separate, related functions combined in one .cpp and one .h file:
- Game_Command.cpp/.h - Handles commands from the user input. (2 files)
- You will also be turning in main.cpp and a Makefile.

1. Class Specifications

Each class and its members are described below by listing its name, the prototypes for the member functions, and the names and types of the member variables. You **MUST** use the prototypes, types, and names in your program as specified here, in the <u>same upper/lower case</u>. **Failure to follow these specifications will result in lost points.**

Cart_Point (10 points)

This class contains two double values, which will be used to represent a set of (x, y) Cartesian coordinates. This class and Cart_Vector(described below) will be used to simplify keeping track of the coordinates of each object in the game, and updating their locations as they move. All data members and functions for this class should be **public**.

Public Members
- double x
  - The x value of the point.
- double y
  - The y value of the point.

Constructors
- The default constructor initializes x and y to 0.0
- Cart_Point(double inputx, double inputy)
  - Sets x and y to inputx and inputy, respectively.

Nonmember Functions
- double cart_distance(Cart_Point p1, Cart_Point p2)
  - Returns the Cartesian (ordinary) distance between p1 and p2.

Nonmember Overloaded Operators (assume p1 and p2 represent two Cart_Point objects, and v1 represents a Cart_Vector object)

- Stream output operator (<<): produces output formatted as (x, y)
  - Example: If p1 has x = 3.14, y = 7.07 then cout << p1 will print (3.14, 7.07)

- Addition operator (+): p1 + v1 returns a **<u>Cart_Point</u>** object with x = p1.x + v1.x and y = p1.y + v1.y
  - Example: If p1 has x = 3, y = 7 and v1 has x=5, y=-2 then this function should make a new **Cart_Point** with x = 8 and y = 5;

- Subtraction operator (-): p1 - p2 returns a **<u>Cart_Vector</u>** object with x = p1.x - p2.x and y = p1.y - p2.y
  - Example: If p1 has x = 3, y = 7 and p2 has x=5, y=-2 then this function should make a new **Cart_Vector** with x = -2 and y = 9;

Cart_Vector (10 points)

This class also contains two double values, but it is used to represent a vector in the real plane (a set of x and y displacements). The overloaded operators allow one to do simple linear-algebra operations to compute where an object's new location should be as it moves around. Some of the overloaded operators and other functions can be member functions, others cannot. All data members and functions for this class should be public.

Public Members

- double x
  - The x displacement value of the vector.
- double y
  - The y displacement value of the vector.

Constructors

- The default constructor that initializes x and y to 0.0
- Cart_Vector(double inputx, double inputy)
  - sets x and y to inputx and inputy, respectively.

Nonmember Overloaded Operators (assume v1 represents a Cart_Vector object and d represents a non-zero double value)

- Multiplication operator (*): v1 * d returns a CartVector object with x = v1.x *d and y = v1.y * d
  - Example: If v1 has x=10 and y=20 and d=5 then this function should make a new **CartVector** with x=50 and y =100.

- Division operator (/): v1 / d returns a Cart_Vector object with x = v1.x / d and y = v1.y / d
  - Example - If v1 has x=10 and y=20 and d=5 then this function should make a new **Cart_Vector** with x=2 and y =4.
  - Dividing by zero should just create v1.

- Stream output operator (<<): produce output formatted as <x, y>
  - Example) If v1 has x = 5.3, y = 2.4 then cout << v1 will print <5.3, 2.4>
  - Notice that this is < and > NOT (and) (like for Cart_Point).

Start by writing the Cart_Point and Cart_Vector classes and a couple of their functions. Add the additional functions one at a time, and test each one. Write a TestCheckpoint1.cpp file with a main function to test them. Therefore, create multiple Cart_Point and Cart_Vector objects in order to test their constructors, and the overloaded operators (<<, +, -, *, /). Getting the overloaded output operator to work soon will make testing the remaining functions more fun.

DO NOT MOVE ON UNTIL YOU ARE 100% SURE YOU UNDERSTAND THIS CHECKPOINT If you do not make sure that these two objects work fully, the rest of your game will NOT work. Use this time to also start to modify the provided makefile to also work for testing Checkpoint 1.

Game_Object (20 points)

This class is the base class for all the objects in the game. It is responsible for the member variables and functions that they **all** have in common. It has the following members:

Private members:
- int id_num;

Protected members:
- CartPoint location;
  - The location of the object.
- char display_code;
  - How the object is represented in the View.
- char state;
- State of the object; more information provided in each derived class.

Public members:
- Game_Object(char in_code, int in_id);
  - Initializes the display_code, id_num, and location to (0,0). It outputs the message: **"Game_Object default constructed"**.
- Game_Object(char in_code, int in_id, CartPoint in_loc);
  - Initializes the display_ code, id_num, and location. It outputs the message: "Game_Object constructed".

- Cart_Point get_location();
  - Returns the location for this object.

- int get_id();
  - Returns the id_num for this object.

- void show_status();
  - Outputs the information contained in this class: display_code, id_num, location. i.e. **"<display_code> with ID <id_num> at location <location>"**. See sample output for exact formatting.

(Notice that later you will be adding a **pure virtual function** called "update()". This will come when you learn about the model).

Gold_Mine (20 points)

This class has a location and an amount of gold. It also has a display_code letter and an id number that are used to help identify the object in the output.

Private Members

- char display_code
  - A code letter to help identify what kind of object is in the output.
  - Initial value should be set to 'G'.
- int id_num
  - An ID number to identify a particular object in the output.
  - Default initial value should be set to 0.
- char state
  - A character identifying the state of the object
  - Initial value is 'f' for "full of gold".
- double amount
  - The amount of gold currently in the mine
  - Initial value should be set to 100.
- Cart_Point location
  - This object's coordinates in the real plane.
  - Cart_Point's default constructor will initialize this to (0.0, 0.0).

Constructors

- The default constructor that initializes the member variables to their initial values listed above.
  - Should print out the message "Gold_Mine default constructed".
- Gold_Mine(int inputId, Cart_Point inputLoc)
  - Initializes the id number to inputId, and the location to inputLoc, and remainder of the member variables to their default initial values.
  - Prints out the message "Gold_Mine constructed".

Public Member Functions

- Cart_Point get_location()
  - Returns a Cart_Point containing the location of the object.
- int get_id()
  - Returns id_num.
- bool is_empty()
  - Returns true if this Gold_Mine contains zero gold. Returns false otherwise.
- double dig_gold(double amount_to_dig = 35.0)
  - If the amount of gold in the mine is greater than or equal to amount_to_dig, it subtracts amount_to_dig from Gold_Mine's amount and returns amount_to_dig. If the amount of gold in the mine is less, it returns the Gold_Mine's current amount, and amount is set to 0. **NOTICE THIS HAS A DEFAULT ARGUMENT VALUE; Chapter 6.7.**
- bool update()

- ○ If the gold mine is empty, it sets the state to 'e' for "empty", change display_code to 'g', prints the message "Gold_Mine (id number) has been depleted." and lastly, returns true. Returns false if it is not depleted.
    - ○ This function shouldn't keep returning true if the Gold_Mine is depleted. It should return true ONLY at the time when the Gold_Mine becomes empty, and return false for later "update()" function calls.
- void show_status()
    - ○ Prints out the status of the object (display_code, id_num, location, and the amount of gold). See the sample output for the format.

CHECKPOINT II
Write the Game_Object and Gold_Mine classes. To test these classes, also write a TestCheckpoint2.cpp. Instantiate multiple objects of these classes in the main function and test out their functions (e.g. update(), dig_gold(), etc.) in order to ensure their proper behavior. For example, call each object's show_status() method after calling their update() method.

DO NOT MOVE ON UNTIL YOU ARE 100% SURE YOU UNDERSTAND THIS CHECKPOINT
If you do not make sure that these Game_Object and Gold_Mine work fully, the rest of your game will NOT work. Use this time to also start to modify the provided makefile to also work for testing Checkpoint 2. DON'T RUSH THROUGH THIS.

Town_Hall (20 points)

A Town_Hall object has a location and an amount of gold. It also has a display_code letter and id number that are used to help identify the object in the output. It has the following members. For clarity, the private members are described first, although normally they are declared after the public members in the code.

- char display_code
  - A code letter to help identify what kind of object this is in the output.
  - Initial value should be set to 't'.
- int id_num
  - An ID number to identify this particular object in the output.
  - Default initial value should be set to 0.
- char state
  - A character identifying the state of the object
  - Initial value is 'o' for "original level".
- double amount
  - The amount of gold currently in the town hall
  - Initial value should be set to 0.
- Cart_Point location
  - This object's coordinates in the real plane.
  - Cart_Point's default constructor will initialize this to (0.0, 0.0).

- The default constructor that initializes the member variables to their initial values listed above.
  - Prints out the message "Town_Hall default constructed".
- Town_Hall(int inputId, Cart_Point inputLoc)
  - Initializes the id number to inputId, and the location to inputLoc, and remainder of the member variables to their default initial values listed above.
  - Prints out the message "Town_Hall constructed".

- Cart_Point get_location()
  - Returns a Cart_Point containing the location of the object.
- int get_id()
  - Returns id_num.
- void deposit_gold(double deposit_amount)
  - adds the amount supplied to the gold in the Town_Hall.
- bool update()
  - If the Town_Hall has gold greater than or equal to 50, it sets the state to 'u' for "Upgraded", change display_code to 'T', prints the message "Town_Hall (id number) has been upgraded." and lastly, returns true. Returns false if it does not have enough gold.

- o This function shouldn't keep returning true if the Town_Hall is upgraded. It should return true ONLY at the time when the Town_Hall gets upgraded, and return false for later "update()" function calls.
- void show_status()
    - o Prints out the status of the object (display_code, id_num, location, and the amount of gold, and a message about whether the Town_Hall has been upgraded or not). See the sample output for the format.

How the Gold_Mine and Town_Hall Behave

These objects also change state, but they do so very simply. The Gold_Mine update function simply checks to see if the mine is empty; if so, it changes the state to 'e' for empty, and signifies a deteriorated mine by changing the display_code letter from 'G' to 'g', and returns true to signify that an event has happened. Similarly, the Town_Hall update function checks to see if the amount of gold on deposit is more than 50; if so, it changes state to 'u' for "upgraded" and changes the display_code letter from 't' to 'T' to show that the now wealthy town has remodeled the building, and returns true to mark this great occasion. They inherit from Game_Object. The constructors should output messages "Gold_Mine default constructed" and "Tone_Hall constructed" as before. The show_status functions so that they output "Town_Hall status:", then call the shadowed Game_Object::show_status() function, and then output the information specific to Town_Hall or Gold Mine. When the Miner objects are created, you should see the proper sequence of messages from the constructors.

Person (15 Points)

This class inherits from Game_Object. It will represent objects that move around and can be commanded to do things. At present, it has only one derived class, Miner. It is responsible for the data and functions for moving around, and provides them as a service to derived classes. Copy the appropriate code from your previous Miner class to your new Person class, and then delete it from Miner.

Public members:
- Person();
- Person (char in_code);
    - o It initializes the speed to 5 and outputs a message: "Person default constructed."
- Person(char in_code, int in_id, Cart_point in_loc)
    - o It invokes Game_Object(in_code, in_id, in_loc) and initializes the speed to 5. It outputs a message: "Person constructed".
- void start_moving(Cart_point dest);
    - o Tells the Person to start moving.
    - o Calls the setup_destination() function.
    - o Sets the state to 'm' for "Moving."
    - o Prints the message "Moving (id) to (destination)." and another message "(display_code)(id): On my way."
- void stop();
    - o Tells the Miner to stop moving or mining.
    - o Sets the state to 's' for "Stopped".
    - o Prints "Stopping (id)." and another message "(display_code)(id): All right."
- void show_status();

- It first calls Game_Object::show_status(), and then outputs the information contained in this class for the moving state and the stopped state: speed, destination, delta

Protected members:
- bool update_location();
    - Updates the object's location while it is moving (See "How the Miner Moves" for details).
    - Prints "(display_code)(id): I'm there!" if Miner has arrived to destination. Prints "(display_code)(id): step…" otherwise.

- void setup_destination(Cart_point dest);
    - Sets up the object to start moving to dest.

Private members:
- double speed;
    - The speed this object travels, expressed as distance per update time unit.
    - Initial value should be 5.

- Cart_Point destination;
    - This object's current destination coordinates in the real plane.
    - Cart_Point's default constructor will initialize this to (0.0, 0.0).
- Cart_Vector delta;
    - Contains the x and y amounts that the object will move on each time unit.

Miner (15 points)

Miner is a type of person. Miner has only the following members. The Miner class represents a simulated human character in the game. It can be told to move to a specified location, and it does so at a certain speed. When it arrives to the location, it stops. It can also be told to start mining a specified Gold_Mine and take the gold to a specified Town_Hall. The Miner object keeps track of the Gold_Mine and the Town_Hall by keeping pointers to the two objects, which allows it to access their locations, extract gold from the mine, and deposit gold in the hall. The behavior of the Miner class is mainly represented in the update() function, and the functions and variables associated with making it move around.

Public members:
- Miner();
    - It invokes Person('M'), and initializes amount to 0, and the mine and home pointers to NULL. As before, it outputs a message: "Miner default constructed."
- Miner(int in_id, Cart_Point in_loc);
    - It invokes Person('M', in_id, in_loc), and initializes amount to 0, and the mine and home pointers to 0. As before, it outputs a message: "Miner constructed."

Public Member Functions
- bool update()
    - Returns true when the state is changed.
    - Updates the status of the object on each time unit (See "How the Miner Behaves" for details).
- void start_mining(Gold_Mine* inputMine, Town_Hall* inputHome)

- Tells the Miner to start mining: move gold from the mine to home, which are specified by pointers to a Gold_Mine and Town_Hall objects.
    - Calls setup_destination() function.
    - Sets the state to 'o' for "Outbound."
    - Prints "Miner (id) mining at Gold_Mine (id) and depositing at Town_Hall (id)." and another message "(display_code)(id): Yes, my lord."
- Cart_Point get_location()
    - Returns a Cart_Point containing the location of the object.
    - It outputs something: "Miner status:", then calls Person::show_status() and then displays the information about the Miner-specific states of inbound, outbound, getting gold, and depositing gold.

Private Members

- char display_code
    - A code letter to help identify what kind of object this is in the output.
    - Initial value should be set to 'M'.
- int id_num
    - An ID number to identify this particular object in the output.
    - Default initial value should be set to 0.
- char state
    - A character identifying the state of the object
    - Initial value is 's' for "stopped".
- double amount
    - The amount of gold currently being carried.
    - Initial value is 0.
- Cart_Point location
    - This object's coordinates in the real plane.
    - Cart_Point's default constructor will initialize this to (0.0, 0.0).
- Gold_Mine* mine
    - A pointer to the Gold_Mine to be mined.
    - Initial value should be 0 (NULL).
- Town_Hall* home
    - A pointer to the Town_Hall where the gold will be deposited.
    - Initial value should be 0 (NULL).

How the Miner Moves

The main function of the program accepts commands from the user. Simulated time is stopped while the user enters commands. The user can command individual objects to move to specified destination coordinates. When the user tells the program to "go", one step of simulated time then happens, and the program calls the update function on every object. The program then pauses to let the user enter more commands, and when the user commands "go" again, another step of simulated time happens, and every object is updated again. So each "go" command corresponds to one "tick" of the clock, one step of the simulated time. The "run" command conveniently makes the program run until an important event happens (to be defined).

An object is commanded to move by calling its start_moving function and supplying the destination. The start_moving function does the following: Call the setup_destination function to save the destination and calculate the delta value. Then set the object in the moving state. The delta value contains the amount that the object's x and y coordinates will change on each update step. We calculate it once, and then apply it on each step. This is the purpose of the overload operators for Cart_Point and Cart_Vector. To calculate the value of delta, use:

delta = (destination – location) * (speed / cart_distance(destination, location))

In other words, the object will move on a straight line, moving a distance equal to its speed on each step. The change in the x and y values of the location on each step are thus proportional to the ratio of the speed to the distance. So the setup_destination function calculates the delta value to be used in the updating steps.

On each step of simulated time, the main routine will call each object's update function. The update function for Miner does a variety of things, but if the object is moving, it calls the update_location function. This function first checks to see if the object is within one step of its destination (see below). If it is, update_location sets the object's location to the destination, prints an "arrived" message, and then returns true to indicate that the object arrived. If the object is not within a step of destination, update_location adds the delta to the location, prints a "moved" messaged, and returns false to indicate that the object has not yet arrived. Thus the object will take a "speed-sized" step on each update "tick" until it gets within one step of the destination, and then on the last step, goes exactly to the destination.

Finally, notice that the user can command all of the Miner objects to move to a destination, and then tell the program to "go", and all of the objects will start moving, and each will stop when it arrives at its own destination. The objects are responsible for themselves!

**An object is within a step of the destination if the absolute value of both the x and y components of fabs(destination - location) are less than or equal to the x and y components of delta (use the fabs() function in math.h library). By checking our distance with very simple computations using the delta value, we don't have to calculate the remaining cart_distance and compare it to the speed using a slow square root function on every update step.

How the Miner Behaves

The behavior of the Miner class is programmed using an approach called a "state machine". A state machine is a system that can be in one of several states and behaves depending on what state it is in. It can either stay in the same state, or change to a different state and behave differently. The neat feature of this approach is that you can easily specify a complicated behavior pattern by simply listing the possible states, and then with each state, describe the input/output behavior of the machine and whether it will change state (See http://en.wikipedia.org/wiki/Finite-state_machine for detail).

In the Miner class, the update function will do something different depending on the state of the Miner object. The state is represented with a simple char variable that contains a code letter for the particular state. A good way to program this is to use a switch statement that

switches on the state variable and has a case for each possible state. In each case, perform the appropriate action for the state, and if needed, change the state by setting the state variable to a different value. Then the next time the update function is called, the Miner will do the appropriate thing for the current state. Thus, the update function contains nothing but a big switch statement.

Here are the states of the Miner, and what the update and show_status functions do for each state (note that update function also does some printing). Generally, the update function should return true whenever the state is changed and return false if it stays in the same state:
- 's' for "stopped."
    - The Miner does nothing and stays in this state.
    - show_status() prints "Stopped."
- 'm' for "moving to a destination"
    - Call update_location to take a step; if the object has arrived, set the state to 's' and return true. Otherwise, stay in the moving state.
    - show_status() prints "Moving at speed (speed) to (destination) <X, Y> amounts on each step are (delta)."
- 'o' for "outbound mining"
    - Call update_location. If it has arrived, set the state to 'g', and return true; otherwise stay in the outbound_ state.
    - show_status() prints "Outbound to a mine."
- 'g' for "getting gold"
    - Call the Gold_Mine's dig_gold function.
    - Print "(display_code)(id): Got (amount) gold."
    - Call setup_destination to prepare to move to the home Town_Hall.
    - Set the state to 'i' and return true.
    - show_status() prints "Getting gold from mine."
- 'i' for "inbound mining"
    - Call update_location. If it has arrived, set the state to 'd', and return true; otherwise stay in the inbound state.
    - show_status() prints "Inbound to home with load: (amount)"
- 'd' for "depositing gold"
    - Print "(display_code)(id): Deposit (amount) of gold."
    - Call deposit_gold function of Town_Hall and set amount to 0.
    - If the Gold_Mine is depleted, set the state to 's' and print "(display_code)(id): More work?" and return true.
    - Otherwise call setup_destination to mine again, state is set to 'o', print "(display_code)(id): Going back for more." and return true.
    - show_status() prints "Depositing gold."

Thus if the Miner is commanded by start_moving to move to a destination, it goes into the moving state, starts moving, and then stops when it arrives and does nothing until commanded again. If the Miner is commanded by start_mining to mine gold and bring it home, the following happens: The start mining function sets the Gold_Mine and Town_Hall pointer member variables, calls setup_destination to prepare to move to the mine, and sets the state to 'o'. The miner goes to the mine, collects some gold, and then calls setup_destination to prepare to move to home. When it gets there, it deposits the gold, sets up the mine as the next destination, and

repeats. When the mine has been emptied, it stops and waits for a new command. Note that it spends a whole update time "tick" in the mine, and in the town hall, before setting out again. So, one time tick passes when it is in the mine, and one time tick passes when it is in the town hall.

The Model-View-Controller (MVC) Pattern

Model (15 points)

The Model is a central component in the MVC pattern and stores all game objects in-memory. Hence, it contains various arrays of pointers to the instances of the GameObject class. Also it offers multiple methods to interact with the Controller and View components. Here, it has the following structure:

Private members:

- int time;
  - the simulation time.

We have a set of arrays of pointers and a variable for the number in each array:

- Game_Object * object__ptrs[10];
- int num_objects;
- Person * person_ptrs[10];
- int num_persons;
- Gold_Mine * mine__ptrs[10];
- int num__mines;
- Town_Hall hall_ptrs[10];
- int num_halls;

Each object will have a pointer in the object _ptrs array and also in the appropriate other array. For example, a Miner object will have a pointer in the object_ptrs array and in the person_ptrs array.

Note: make the copy constructor private to prevent a Model object from being copied

Public members:

- Model();
  - It initializes the time to 0, then creates the objects using **new**, and stores the pointers to them in the arrays as follows:
  - The list shows the object type, its id number, initial location, and subscript in object_ptrs, and the subscript in the other array.
  - Miner 1 (5, 1), object _ptrs[0], person_ptrs[0]
  - Miner 2 (10, 1), object _ptrs[1], person_ptrs[1]
  - Gold_Mine 1 (1, 20), object_ptrs[2], mine_ptrs[0]
  - Gold_Mine 2 (10, 20), object_ptrs [3], mine_ptrs [1]
  - Town_Hall - default constructed - object_ptrs[4], hall_ptrs[0]
  - Set num_objects to 5, num_persons to 2, num_mines to 2, num_halls to 1;
  - Finally, output a message: "Model default constructed";
- ~Model();

○ the destructor deletes each object, and outputs a message: "Model destructed."

Note: For purposes of demonstration, add to Game_Object a destructor function that does nothing except output a message: "Game__Object destructed." Define a similar ones for Person, Miner, Gold_Mine, and Town_Hall. This is so that you can see the order of destructor calls for an object.

There are three functions that provide a lookup and validation services to the main program (Controller). They return a pointer to the object identified by an id number, depending on the type of object we are interested in. The functions search the appropriate array for an object matching the supplied id, and either return the pointer if found, or 0 if not.

- Person * get_Person_ptr(int id);

- Gold_Mine * get_Gold_Mine_ptr(int id);

- Town_Hall * get_Town_Hall_ptr(int id);

- bool update();

  ○ It provides a service to the main program. It increments the time, and iterates through the object_ptrs array and calls update() for each object. Since Game_Object::update() will be made virtual, this will update each object.

- Model::update()

  ○ returns true if any one of the Game_Object::update() calls returned true.

- void display(View &view);

  ○ Likewise it provides a service to the main program. It outputs the time, and generates the view display for all of the Game_Objects.

- void show_status();

  ○ It outputs the status of all of the Game_Objects by calling their show_status() function.

Implement the polymorphism in the class hierarchy as follows:

- Declare bool update() to be a pure virtual function in Game_Object. This makes Person an abstract base class and ensures that each of the derived classes will have defined an update function, or we get a linker error to tell us we have left it out.

- Declare show_status() to be virtual in Game_Object.

- Define the start_mining() function in the Person class, make it virtual, and have it do nothing but output a message: "Sorry, I can't work a mine."

- Important: Make the destructors in Game_Object and Person virtual.

Your main function and its subfunctions now can be radically simplified because they are no longer responsible for keeping track of all the objects or how many of them of what kinds there are. Remove all the declarations of Miners, Town_Halls, and Gold_Mines from main, and replace them with declaring a single Model object. Each command function should be given just the Model object (by reference), as in:

- void do_move_command(Model& model);

The command functions should use the Model member functions like get_Gold_Mine_ptr to check that an input id number is valid and get a pointer for the object if it is. The

do_go_command and do_run_command functions can just call Model::update() to update all the objects, and Model::display can be called to display the current view of the game. Move all these do_***_command functions into a separate .h and .cpp file.

Add a new command to list the status of all the objects using the show_status service of the Model:

- l - (lower-case letter L) -list - output the status of all objects

Your program should work as before. You should be able to command the Miners to move around and work, and list the status of all the objects. When the program terminates, you should see the correct sequence of destructor messages.

Game_Command.cpp & Game_Command.h

(15 points)

The Game_Command represents the Controller of the MVC pattern and provides multiple functions that interpret user input in order to perform the appropriate actions.

You should create a set of functions that can be called from main to handling the processing of user provided commands.

Here is a description of the commands and their input values:
- g1 is a Gold_Mine, ID number is 1, initial location is (1, 20).
- g2 is a Gold_Mine, ID number is 2, initial location is (10, 20).
- m1 is a Miner, ID number is 1, initial location is (1, 1).
- m2 is a Miner, ID number is 2, initial location is (10, 1).
- t1 is a Town_Hall, ID number is 1, location is (5, 0).

Here is a description of the commands and their input values:
- **m** ID x y
    - "move": command Miner ID to move to location (x, y)
- **w** ID1 ID2 ID3
    - "work a mine": command Miner ID1 to start mining at Gold_Mine ID2 and deposit the gold at Town_Hall ID3.
- **s** ID
    - "stop": command Miner ID to stop doing whatever it is doing
- **g**
    - "go": advance one time step by updating each object once.
- **r**
    - "run": advance one time step and update each object, and repeat until either the update function returns true for at least one of the objects, or 5 time steps have been done.
- **q**
    - "quit": terminate the program

You do NOT need to do error checking. This will be done in PA4.
You must have a separate command-handling function for each command that collects the input required for the command and calls the appropriate object member functions. We recommend using the switch statement to pick out the function for each command; this is the simplest and

18

cleanest way to do this sort of program branching. For example, to handle the "move" command, the case would look like this:

```
        case 'm':
                do_move_command(m1, m2);
                break;
```

All input for this program must be done with the stream input operator >>.

Important: *Your program must "echo" each command to confirm it and to provide a record in the output of what the input command was. For example, if the user enters "m 1 10 15" the program should output something like "moving Miner 1 to (10, 15)". This way, if output redirection is used to record the behavior of the program, the output file will contain a record of the input. If there is an error in the input, there should be an informative error message, but your program is not responsible for trying to output an exact copy of the erroneous input. So the echo does not have to be present or complete if there is an error in the input.*

CHECKPOINT IV

Implement the Model and the GameCommand.cpp/.h and add them to your program. You now should be able to actively have a user manually enter commands and have them play your game. Write a TestCheckpoint4.cpp file and in the main function command the Miner to mine, dig, etc., and list the status of all game objects. Make sure you comment out parts that reference View as that does not exist yet. When the program terminates, you should see the correct sequence of destructor messages.

DO NOT MOVE ON UNTIL YOU ARE 100% SURE YOU UNDERSTAND THIS CHECKPOINT If you do not make sure that these Model and GameCommands work fully, the rest of your game will NOT work. Use this time to also start to modify the provided makefile to also work for testing Checkpoint 4. DON'T RUSH THROUGH THIS.

View (10 points)

Thanks to inheritance, we can easily add a better display using simple "ASCII graphics." This display will be like a game board – a grid of squares. Each object will be plotted in the grid corresponding to its position in the plane. As the object moves around its position on the grid will be changed. The object will be identified on the board by its display_code letter and its id_num value. **We will be assuming that the id_nums are all one digit in size at this time.** We will provide sample output to show what the display should look like.

The code for this display will be encapsulated in a class of object called a "View" whose function is to provide a view of some objects. The View object has a member function plot() that puts each object into display grid; it will ask Game_Object to provide the two characters to identify itself. Member function draw() will output the completed display grid, and clear() will reset the grid to empty in preparation for a new round of plotting.

In class Game_Object, add the following public member function:

- void drawself(char * ptr);.

  - The function puts the display_code at the character pointed to by ptr, and then the ASCII character for the id_num in the next character.

Write the View class with its own .h and .cpp file to have the following members.

Constant defined in the header file

- const int view_maxsize = 20;

  - the maximum size of the grid array

Private members:

- int size;

  - the current size of the grid to be displayed; not all of the grid array will be displayed in this project.

- double scale;

  - the distance represented by each cell of the grid

- Cart_Point origin;

  - the coordinates of the lower left-hand corner of the grid

- char grid[view_maxsize][view_maxsize][2];

  - an array to hold the characters that make up the display grid.

- bool get_subscripts(int &ix, int &iy, Cart_Point location);

  - This function calculates the column and row subscripts of the grid array that correspond to the supplied location. Note that the x and y values corresponding to the subscripts can be calculated by (location - origin) / scale. Assign these to integers to truncate the fractional part to get the integer subscripts, which are returned in the two reference parameters. The function returns true if the subscripts are valid, that is within the size of the grid being displayed. If not, the function prints a message: "An object is outside the display" and returns false.

Public members:

- View()

- ○ It sets the size to 11, the scale to 2, and lets the origin default to (0, 0). No constructor output message is needed.
- void clear();
  - ○ It sets all the cells of the grid to the background pattern shown in the sample output.
- void plot(Game_Object * ptr);
  - ○ It plots the pointed-to object in the proper cell of the grid. It calls get_subscripts and if the subscripts are valid, it calls drawself() for the object to insert the appropriate characters in the cell of the grid. However, if there is already an object plotted in that cell of the grid, the characters are replaced with '*' and ' ' to indicate that there is more than one object in that cell of the grid.

Tip about base class pointers. C++ normally refuses to convert one pointer type to another. But it will allow a conversion from a Derived class pointer to a Base class pointer (upcasting). This allows you to plot all kinds of Game_Objects with a single function.

Tip about CIC++ arrays: If a is a three-dimensional array, then a[i][j][k] is the i, j, k element in the array, and a[i][j] is a pointer to a one-dimensional array starting a[i][j][0].

- void draw();
  - ○ outputs the grid array to produce the display like that shown in the sample output The size, scale, and origin are printed first, then each row and column, for the current size of the display. Note that the grid is plotted like a normal graph: larger x values are to the right, and larger y values are at the top. The x and y axes are labeled with values for every alternate column and row. Use the output stream formatting facilities to save the format settings, set them for neat output of the axis labels on the grid, and then restore them to their original settings. Specifications: Allow two characters for each numeric value of the axis labels, with no decimal points. The axis labels will be out of alignment and rounded off if their values cannot be represented well in two characters. This distortion is acceptable in the name of simplicity for this project

Now modify your main program to declare a view object. Where you previously did a show_status() call for each object, replace that code with first a call to the View object's clear() function, then call the plot function using each object, and then the draw() function. Now you should have the graphical display. Thanks to the inheritance from Game_Object, the display works for all three types of specific game objects.

Overall Structure of the Program (main.cpp)

(50 points for behavioral tests; update will be posted on Blackboard regarding this requirement and sample output)

The main program should include a loop that reads a command, executes it, and then asks for another command. **You do NOT have to detect errors or bad input. PA4 will add these features.**

The program will declare three Miner objects, four Gold_Mine object, and Town_Hall objects as outlined in the Model class description. These ten objects will persist throughout the execution of the program.

The program starts by displaying the current time and the status of each object using the show status function. The main function then asks for a command and the user inputs a single character for the command, and main calls a function for the appropriate command. The function for the command inputs requests any required additional information from the user, and then carries out the command. If the user entered the "go" or "run" commands, the program repeats the main loop by displaying the time and current status and prompting for a new command. Otherwise, it continues to prompt for new commands. This enables the user to command more than one object to move before starting the simulation running again.

Additional Specifications

You MUST:
- Make use of the classes and their members; you may not write non-object oriented code to do things that the classes can do.
- Declare function prototypes for the functions that are not part of a class, such as those called by main(), and list the function definitions after main. This will improve the readability of your program.
- Make .h and .cpp files for each of your class with data fields and member functions as specified above.
- Make sure your code compiles!

Programming Guideline

Unless you have so much experience that you shouldn't be in this course, trying to write this program all at once is the **hard** way to do it! Objected oriented programming is easy to do in chunks! That's the idea! The individual classes can be written and tested pretty much one at a time, and a piece at a time besides. Here's how to write this program a chunk at a time:

1.   Start by writing the Cart_Point and Cart_Vector classes and a couple of their functions. Write a trivial version of main to test them by creating one of each and doing things with them. Add the additional functions one at a time, and test each one. Getting the overloaded output operator to work soon will make testing the remaining functions more fun.

2. Start on writing the Miner class, but write only the constructors and a partial form of the show_status function; leave everything else out. Write another trivial testing main; create a Miner object, and a show_status _function. It should display the correct initial state of the object to verify both the constructors and the show_status function. Then, add the start_moving, setup_destination function, and update_location functions, and the 's' and 'm' parts of the update and show_status functions. Change your trivial main function to call the start_moving function on your one Miner object. This should show its status, call its update function, and show its status again - it should be in a different location! Check that amount moved on the step is correct. Put in a few more calls of update and see if the Miner stops like it should.

3. Write the Gold_Mine and Town_Hall classes, add the Miner's start_mining function, and the corresponding code to the Miner update function. Add to your testing main function a declaration of a Gold_Mine and Town_Hall object, and call start_mining, then update and show_status on the Miner multiple times. You should see the Miner shuttle between the mine and the town hall.

4. Write the command loop for main(), and put the whole program together, and test it by making the objects all move around. See if you can have the two miners work the two mines in various combinations at the same time. Be sure to put in the input error checking before you do a lot of testing. Check each command with garbage input. Your program should be "bullet-proof"; no matter what the user types, it keeps going. Some clumsiness and stumbling is OK, as long as it eventually is ready to receive a new command.

Makefile

Makefiles automate and facilitate the compilation process of software projects that consists of a large number of source code (.cpp) files (such as PA3). Instead of specifying each .cpp file in the compilation process (g++) of PA3, we will create makefiles to automate the compilation process and the achievement of the checkpoints.

In general, a makefile consists of multiple named targets ("rules") that can depend on each other. Every target has an associated action that is usually a UNIX command, such as g++. The structure of a named target looks as follows:
target_name : dependencies
        command_to_execute

IMPORTANT!
The command_to_execute line MUST be indented using a Tab space. This can be achieved by using the Tab key on your keyboard.

The UNIX make command interprets makefiles by executing the targets and their associated actions in the order of the targets' dependencies. Per default, the make command executes a

makefile that is literally called Makefile (case-sensitive!). In case of naming your Makefile differently (e.g. Makefile_Checkpoint1), then you must use the –f option for the make command. Also check out the manual page of the make command (man make).

To demonstrate makefiles, we provide a makefile for Checkpoint I. Let's name the makefile Makefile_Checkpoint1.

```
# in EC327, we use the g++ compiler
# therefore, we define the GCC variable
GCC = g++

# a target to compile the Checkpoint1 which depends on all object-files
# and which links all object-files into an executable
Checkpoint1: TestCheckpoint1.o CartPoint.o CartVector.o
    $(GCC) TestCheckpoint1.o CartPoint.o CartVector.o -o Checkpoint1

# a target to compile the TestCheckpoint1.cpp into an object-file
TestCheckpoint1.o: TestCheckpoint1.cpp
    $(GCC) -c TestCheckpoint1.cpp

# a target to compile the CartPoint.cpp into an object-file
CartPoint.o: CartPoint.cpp
    $(GCC) -c CartPoint.cpp

# a target to compile the CartVector.cpp into an object-file
CartVector.o: CartVector.cpp
    $(GCC) -c CartVector.cpp

# a target to delete all object-files and executables
clean:
    rm TestCheckpoint1.o CartPoint.o CartVector.o Checkpoint1
```

As demonstrated, makefiles only support single-line comments that start with the '#' character. Furthermore, makefiles support the definition of variables, such as GCC, which get a specific value assigned. In our case, the GCC variable holds the value g++. You can access the value of the variables using the $() notation. For example, $(GCC) returns the value of the GCC variable, which is g++.

In the file Makefile_Checkpoint1 we specify five targets (denoted in bold text). The Checkpoint1 target depends on three targets, namely TestCheckpoint1.o, CartPoint.o, and CartVector.o. Each *.o target depends on the .cpp file, which should be compiled to an object file (using the -c option of the g++ compiler). We define those dependencies in order to avoid re-compilation of .cpp files that have not been changed during the last compilation process. We also define one target clean, which deletes all object-files and the executable Checkpoint1 (using the rm UNIX command).

Per default, the make command executes the first target, which is in our case Checkpoint1. You can, however, instruct what target to execute by passing the name of the target to the make

command. For example, the following command will execute the clean target of the Makefile_Checkpoint1 makefile.
make –f Makefile_Checkpoint1 clean

The EC327 staff will provide a final makefile for PA3. However, it is your task to formulate the makefiles for each checkpoint.

For further information on makefiles, please consult https://www.gnu.org/software/make/

Submission

Please use the file names **Cart_Point.h, Cart_Point.cpp, Cart_Vector.h, Cart_Vector.cpp, Gold_Mine.h, Gold_Mine.cpp, Town_Hall.h, Town_Hall.cpp, Miner.h, Miner.cpp, and main.cpp**. Put all your cpp files in a folder named <your username>_PA3 (e.g., dougd_PA3), zip it, and submit a single file (e.g., dougd_PA3.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.