**1.**

**doSomething()**

doSomething is pure virtual because all actors can “doSomething” (except for rocks and food, but since doSomething is void anyway, I can just say it does nothing), and most actors do things in a different way.

**Pebbles** and **Food** do nothing, so their function is blank.

**Anthill** subtracts energy (by calling getEnergy() and updateEnergy), eats food (using the pickupandEatfood function), checks energy level, and if the energy level is high enough, it allocates a new Ant (and figures out the ant’s ID by using a switch statement) and pushes it into the stl object using the addActor function, increasing the score of the colony.

**Pheromone** subtracts strength of it per turn using getEnergy() and updateEnergy.

**WaterPool** and **Poison** both call on StudentWorld functions – specifically **stunAllStunnableAt** and **poisonAllPoisonableAt** (which themselves call **getStunned && getPoisoned**, which are virtual functions (due to the fact that not all objects are capable of being stunned or poisoned). getPoisoned uses getEnergy and updateEnergy to affect hitpoints, while getStunned just sets the m\_stun private member to 2 )**,** which iterates through the stl objects and stuns/poisons all actors capable of being stunned or poisoned

**Insect** subtracts hitpoints (using getEnergy and updateEnergy) and checks if something is dead – if it is, it drops food onto the STL array by calling a StudentWorld function

**Ant, Grasshopper, AdultGrasshopper** all call on Insect doSomething(). Grasshopper and Adult Grasshopper have very similar codes, but are separated since a) there are some differences b) I couldn’t figure out how to fix it on a time crunch.

**Grasshopper** calls Insect::doSomething, checks if dead, if so, it immediately returns. It then checks to see if it’s stunned – if so, again, immediate return. **AdultGrasshopper** at this point, randomly bites an enemy using the studentWorld function **biteEnemy()** (which iterates through the stl array checking if there’s an enemy on the grasshopper’s square, and if so, it calls a **getbitten()** function that uses updateEnergy and getEnergy to up). **BabyGrasshopper** at this point, checks to see if its hitpoints are high enough, and if so, it sets itself to dead, adds a food object to its position, and allocates a new AdultGrasshopper, which gets pushed into the STL array.

**updateEnergy()&& getEnergy()**

While this is defined for all Actors, it automatically returns zero. It’s virtual so that I can adjust the energy levels for the Energy class (by subtracting or adding from a private energy member) and view the energy levels for the EnergyHolders while still being able to call Energy for actors

**isDead()**

For the Actor class, this automatically returns false. For **EnergyHolders,** this checks the amount of energy it has, and if it’s zero or less, this will return true

**blocksMovement()**

is a virtual function that will automatically return false, since only one object blocks movement. This is here so we can actually call blocksMovement on all Actors, in order to see which object (**Pebble**) is an obstacle.

**getBitten()**

is virtual, since not all objects can get bitten and the objects that *can* get bitten/stunned/poisoned do so in different ways – the adult grasshopper will automatically bite back if bitten.

**isEdible()**

is virtual since not all Actors can be eaten, but the function still needs to be called on all Actor pointers to figure out what *can* be eaten.

**isPhermone()&& isMyPheromone**

is virtual – it will automatically return false for all except the Pheromone class, in which case it’ll check to see if the pheromone belongs to a certain colony, but the function still needs to be called on all Actor pointers to figure out what is a pheromone.

**isDangerous() && isEnemy()**

has random values in the base Actor class, since not everything is going to be dangerous and not everything is going to be an enemy.

**IncreaseStrength()**

does nothing here, but for Pheromones, it increases the strength of a pheromone, and is a virtual public function since I couldn’t figure out how to access certain variables and deemed this a better option than making a bunch of private members

**isMyHill(int colony)**

is really only for anthill, and checks to see if a colony is the same colony as the hill’s.

**returnStun() && lessStun()**

are essentially only for the Insect class, since only insects can be stunned. All insects are stunned in the same way so this is not a virtual function – these were essentially made to be able to access the private member variables in Insect’s derived classes.

**becomesFoodUponDeath()**

is selfexplanatory. It’s function that states whether an object becomes food upon death, and simply returns true or false depending on the class; It is *virtual* since not all objects actually do become food upon death – **Food** and **Pheromone** for example.

**distance() && setDistance**

are really only for the grasshopper class so I could try and access the private distance member. Anyway, distance returns the distance member, while setDistance allows me to alter it. So, not virtual, and borne out of the fact that writing it more concisely would have taken more time.

**init()**

initialize the studentWorld:

Create compilers, compile student world, load the field.textFile, and iterate through the Field STL object to get its contents and therefore allocate corresponding items into the

**move()**

move objects in the stl arrays, and deletes dynamically allocated objects.

iterates through the entire STL array, while checking if the objects are dead – if so, then the object gets erased.

iterates through the STL object a second time, calling the doSomething() function

**cleanup()/~StudentWorld()**

iterates through the entire array deleting everything;

**canMoveTo(int x, int y)**

searches through the vector at position x, y for a rock (thus calling the blocksMovement() function while iterating through the vector)

**addActor(Actor\* a)**

adds an actor by pushing it into the stl object in a position that a indicates, due to private x and y variables. All actors are treated the same way, so no need to be virtual

**getEdibleAt(int x, int y)**

returns an Actor pointer that refers to an edible Actor. This is done by iterating through the array and using the **isEdible()** function to check whether something is in fact edible. All actors are treated the same way, so no need to be virtual

**getPheromoneAt(int x, int y, int colony)**

If a pheromone of the indicated colony is at x,y, return a pointer to it. This is done by iterating through the array and using the **isPheromone(int colony)** function to check whether something is in fact a viable pheromone. All actors are treated the same way, so no need to be virtual

**isEnemyAt(int x, int y, int colony)&&  isDangerAt(int x, int y, int colony) const;**

Iterates through the array using the isEnemyAt and isDangerAt actor functions. All actors are treated the same way, so no need to be virtual.

**2.** As far as I could tell, the grasshoppers, pebbles, food, and are working properly – mostly, at least. The triggerable actor functions don’t take into account getting restunned or repoisoned, and the adult grasshopper is currently capable of being stunned. There is currently no winner. The ants also don’t appear to be moving. This is still better than when I tried looping through my simple instructor to make my ant take 10 commands before returning, which caused absolutely all objects to stop moving. This leaves me unsure if my pheromone class is working properly, since, as the ant doesn’t move, I can’t tell if it’s leaving any pheromones. I’m not quite sure if the USCant data file is exceedingly stupid or if I messed up.

**3.** In the simple iterator, I assumed that command does not do this. I also assumed that the grasshoppers had no maximum hitpoints, and could theoretically simply keep increasing until there was no more food left in the field.

**4.** Had I started two days earlier, I might have been able to test everything. Alas, I am terribly lacking in time management skills.

However, if I *had* started two days earlier, here is how I would presumably test my classes:

StudentWorld: the game actually loading and playing is a fairly good test. It’d also be good to inspect the screen in order to check that all graphic objects have been loaded properly, and to track doSomething to make sure it doesn’t return a nullptr. I’d also print out the number of ticks and number of ants *very clearly* to make sure that those actually increment.

Actors:

Pebbles: One idea could be to make a data field.txt with one grasshopper absolutely surrounded by rocks. If the grasshopper didn’t move at all, that would be proof that the pebbles do in fact successfully block the paths. I suppose I could also make a field.txt file made up of only pebbles, to make sure it was really do *nothing* when it was called to do something.

EnergyHolders need to be checked to see if actors really do pickup food, if the actors become food, and if the amount of energy changes. All EnergyHolders would need to be checked for this. One easy way to see if the actors become food would be to simply create a grasshopper in an emptyfield and watch it starve to death, to make sure that the energy *is* getting updated (if it were not, the hitpoints wouldn’t be affected) and that the grasshopper actually does become food.

Food: Food needs to be checked if 1) it can actually be eaten 2)if so, does it increase hitpoints 3) does it deallocate properly. To make sure the food is getting picked up, it would be nice to make a field.txt of a grasshopper surrounded by food (of a lower energy than required). I’d be able to see the food objects being deallocated if this worked. A more accurate way would be to search through the array while writing out a function that allowed me to see how much energy each object had, to make sure the energy was getting transferred properly. Currently, while it’s difficult for me to see if the food objects are, in fact, being deallocated, the baby grasshoppers do appear to be molting into adult grasshoppers, so they *are* picking up and eating food.

Anthills: If I wanted to make sure the Anthill was being correctly deallocated, I could make its hitpoints equal a suitably low number and wait until the Anthill was dead. if I wanted to make sure the Anthill was eating without having to rely on ants, I could probably drop a suitably large amount of food on it and wait to see if the food object disappears, and if so, if the Anthill produced new Ants.

Pheromones: will be difficult, since they don’t really leave traces. I suppose you could check if the strength decreases by making a field.txt file of only pheromones, so you could check to see if the strength of the pheromone was dying.

Triggerable Actor: is a fairly simple class, so whether or not it works is dependent on:

Poison/WaterPool: Since these are quite similar classes, I could probably set a couple of breakpoints on their implementations and move through while writing a function that returns the m\_stun and m\_poison private member variables I had in place. Alternatively, I could make a field.txt that consisted only of poison or waterpools, with one grasshopper around. I could then check if the grasshopper was getting repeatedly poisoned or stunned.

Insect: One easy way to see if the insects become food would be to be to create an insect in an empty field and watch it starve to death, to make sure that the energy *is* getting updated (if it were not, the hitpoints wouldn’t be affected) and that the grasshopper actually does become food It’s also important to watch the screen closely to make sure the insects aren’t moving out of turn (this can be done by increasing the amount of turns the insects spend asleep, so this can be *actually* watched).

Ant: This is difficult purely because of the sole amount of *code* that goes into the ant class. So first you’d have to check to see if it could starve (empty field), and then you’d have to see if it could actually pick up food – perhaps a field with only two spaces to move and one food object that is precisely within that ant’s ability to pick up. If the food object disappears, and you know the ant isn’t hungry, then it must have picked up the food object.

Grasshopper: this really is essentially an implementation for BabyGrasshopper, so I’ll cover both of these in this paragraph. Baby grasshopper needs to be able to starve and molt, which can be checked by creating a completely empty field and a field full of food respectively. BabyGrasshopper also needs to be stunnable and poisonable, which can be proven being creating a field full of water and a field full of poison.

AdultGrasshopper needs to be able to bite back and jump. To check to see if the adult Grasshopper actually bites back, you could create an incredibly cramped field surrounded by stones, leaving the adultGrasshopper utterly nowhere to go except to squares with babyGrasshoppers around. if you increase health accordingly (so you can be very sure that the babyGrasshopper didn’t die of starvation), you could probably watch the babyGrasshopper get bitten to death. Jump is harder to prove. I’d assume you’d have to severely slow down the Grasshopper and then watch to see if the grasshopper actually does take a leap that is within ten squares.