Project 3 Report

**Class StudentWorld : Public GameWorld -**

None of these functions are virtual because StudentWorld has no derived classes.

StudentWorld: Constructs the StudentWorld object, initializing elapsed ticks (m\_ticks) to 0, maxAnt (the winning ant) to 0, and leaves all the vectors used empty.

int init: Performs the initial set up of the world. First the field text file is loaded. If no error occurs, the field text is traversed with two nested for loops, and whenever a field item is found, it is constructed and its pointer added to the 2D array of vectors that hold the actors (except for anthills, whose positions are stored to possibly be used later). Next, depending on the number of ants competing, the compilers are constructed. If an error occurs, all compilers created thus far are deleted. Each time a compiler is successfully created, its pointer is added to a vector, an initial score of 0 is added to another vector, and the corresponding anthill is constructed and pushed back onto the 2D array of vectors. If no errors have occurred, the signal to start the game is returned.

int move: Runs the simulation. First the number of elapsed ticks is increased. Then each actor is asked to doSomething() using nested for loops. Insects do something before other actors, otherwise sometimes objects will not act correctly (e.g. grasshoppers not getting stunned). Then all the dead actors are removed, using nested for loops again. The game text is set and displayed. Once 2000 ticks is reached, if the leading ant meets the victory conditions, then it is set as the winner. Otherwise, no winner is declared.

void cleanUp: Deletes every dynamically allocated object via nested for loops for all the actors and another for loop to delete all the compilers that were stored.

~StudentWorld: Calls cleanUp();

bool canMove: Checks if an x, y coordinate is in bounds, and if so, will ask each actor there if they can be moved through. If ever receiving false, it will return false. Insects use this whenever they try to move, because they cannot interact with other actors without going through the world.

void poison: Attempts to poison every actor at an x, y coordinate. Poison objects need to ask the world to poison everything in their space, because they cannot interact with other actors without going through the world.

void stun: Attempts to stun every actor at an x, y coordinate. Pool objects need to ask the world to stun everything in their space, because they cannot interact with other actors without going through the world.

void bite: Attempts to bite an enemy of a certain colony at an x, y coordinate that is not the actor who called the function. Adult grasshoppers and ants need to ask the world to bite some enemy in their space, because they cannot interact with other actors without going through the world.

bool isFoodPresent: Checks if any actors at an x, y coordinate are food. Insects and anthills need to ask the world when there is food where they are at, because they cannot interact with other actors without going through the world.

bool isEnemyInsectPresent: Checks if any actors at an x, y coordinate are enemies of a certain colony. If performing the check for grasshoppers, two enemies need to be found, as the grasshopper who called the function will be found as enemy of grasshoppers. Adult grasshoppers and ants need to ask the world if there are enemies in their space if they want to bite something, because they cannot interact with other actors without going through the world.

bool isPheromonePresent: Checks if any pheromone is at an x, y coordinate. Ants need to ask the world if there is pheromone where they are looking, because they cannot interact with other actors without going through the world.

bool isAnthillPresent: Checks if a certain colony’s anthill is at an x, y coordinate. Ants need to ask the world if their anthill is where they are at, because they cannot interact with other actors without going through the world.

bool detectDanger: Checks if any actors at an x, y coordinate are dangerous to a certain colony. Ants need to ask the world if there is anything dangerous in front of them, because they cannot interact with other actors without going through the world.

int addFood: If a positive value is passed, and there is already a food object, then the object’s health is increased by the value’s amount. If there is not already a food object, one is created. The negative of the value passed is returned, but this is never used in this situation. If a negative value is passed, then the food’s health is decreased by that amount, to a minimum of zero. The amount by which the food’s health was decreased is returned. Insects use this when they die or when they want to pick up food/eat (as do anthills), because they cannot interact with other actors without going through the world.

void addAdultGrasshopper: Adds an adultGrasshopper with the specified arguments for construction. Baby grasshoppers use this when reaching 1600 health, because they cannot interact with other actors without going through the world

void addAnt: Adds an Ant with the specified arguments for construction. Anthills use this when they have 2000 or more health to birth another ant, because they cannot interact with other actors without going through the world.

void addPheromone: Adds pheromone with the specified arguments for construction. If there is already a pheromone of the specified colony at the specified x, y coordinate, the pheromone’s health is increased by 256 up to a max of 768. Ants use this when they want to emit pheromone, because they cannot interact with other actors without going through the world.

void incrementScore: Add 1 to the specified colony’s score. Anthills use this whenever they produce another ant. This allows the world to determine the winner.

int getMaxAnt: Check if there is a new colony in the lead, and return the leading colony’s number. This is used to create the correct display text and allows the world to determine the winner.

std::string getDisplayText: Using a stringstream, the display text string is created by first printing out the amount of ticks remaining, then how many ants are competing and printing out their names and score. If a competing ant is also the maxAnt, and meets the victory conditions, then its name is followed by a ‘\*’ in the display text. This function allows the world to display simulation information.

**Class Actor -**

Actor: Constructs an actor with the specified arguments.

virtual void doSomething: I chose to define this as virtual because not all actors doSomething in the same way, but I wanted to create some default behavior for actors (do nothing).

virtual bool isDead: I chose to define this as virtual because not all actors can die, but I wanted some default dying behavior (never dying).

virtual bool canMoveThrough: I chose to define this as virtual because it is to distinguish pebbles from the other actors. All other actors can be moved through, so the function returns true by default.

virtual int callCount: I chose to define this as virtual because callCount only comes into play for Insects, so this provides default behavior of callCount always being less than the number of elapsed ticks.

virtual void poison: I chose to define this as virtual because not all actors can be poisoned, so this provides the default behavior of doing nothing when poisoned.

virtual void stun: I chose to define this as virtual because not all actors can be stunned, so this provides the default behavior of doing nothing when stunned.

virtual bool isDangerous: I chose to define this as virtual because not all actors are dangerous, so this provides the default behavior of not being dangerous.

virtual bool isEnemy: I chose to define this as virtual because not all actors can be enemies, so this provides the default behavior of not being an enemy.

virtual bool isPheromone: I chose to define this as virtual because it is to distinguish Pheromones from the other actors, so the function returns false by default.

virtual bool isMyPheromone: I chose to define this as virtual for the same reason as above. This function is needed whenever new pheromone is emitted.

virtual bool isMyAnthill: I chose to define this as virtual because it is distinguish anthills from other anthills and actors, so the function returns false by default.

virtual bool isFood: I chose to define this as virtual because it distinguishes food from other actors, so it returns false by default.

virtual bool isInsect: I chose to define this function as virtual because not all actors are insects. This function is used in StudentWorld::move to make sure insects do something before everything else.

StudentWorld\* world: I chose to define this as non-virtual because all actors access their world in the same way. This function is needed for any interactions with the simulation state.

virtual ~Actor(): Since this is a base class, it needs a virtual destructor.

**Class Pebble : Public Actor -**

Pebble: Constructs a pebble at the specified coordinate.

bool canMoveThrough: I chose to define this as non-virtual because Pebble does not serve as a base class. Pebbles cannot be moved through, so this returns false. This allows pebbles to be distinguished.

**Class energyHolder : Public Actor -**

energyHolder: Constructs an energyHolder with the specified arguments. energyHolders have health and a colony number.

bool isDead: I chose to define this as non-virtual because all energyHolders die in the same way, if their health is 0 or less. This function is needed for several interactions, including removing dead actors, and determining whether or not some action will be performed on an actor.

int changeHealthBy: I chose to define this as non-virtual because all energyHolders have their health changed in the same way. This function takes in how much the health is supposed to change by and returns how much health was actually subtracted from the energyHolder, which is needed for whenever food is picked up.

int health: I chose to define this as non-virtual because all energyHolders report their health in the same way, by returning their hp. This function is needed for anything that requires health to be checked, such as turning into an adult grasshopper or giving birth to an ant.

int colonyNumber: I chose to define this as non-virtual because all energyHolders report their colonyNumber the same way, by returning their colonyNumber. This function is used by anthills, ants, and pheromones.

void changeColonyNumber: I chose to define this as non-virtual because all energyHolders change their colony number in the same way, by replacing their member variable value with whatever is taken in. This is used in constructing anthills, ants, and pheromones.

virtual ~energyHolder: Since energyHolder is a base class, it needs a virtual destructor.

**Class Food : Public energyHolder –**

Food: Constructs food with the specified arguments.

bool isFood: I chose to define this as non-virtual because Food does not serve as a base class. This returns true and serves to distinguish food from other actors.

**Class Pheromone : Public energyHolder –**

Pheromone: Constructs pheromone with the specified arguments.

void doSomething: I chose to define this as non-virtual because Pheromone does not serve as a base class. This decrements the pheromone’s health each tick.

bool isPheromone: I chose to define this as non-virtual because Pheromone does not serve as a base class. This returns true and serves to distinguish pheromone from other actors.

bool isMyPheromone: I chose to define this as non-virtual because Pheromone does not serve as a base class. This returns true if the number taken in and the colony number match. This is used when ants emit pheromone.

**Class Anthill : Public energyHolder –**

Anthill: Constructs anthill with the specified arguments. Anthills have an associated compiler.

void doSomething: I chose to define this as non-virtual because Anthill does not serve as a base class. This function decrements the anthill’s health and checks if it is dead. If it is not dead, it will check if there is food on its square, and will eat if there is. Otherwise, it will check if it has enough health to birth a new ant.

bool isMyAnthill: I chose to define this as non-virtual because Anthill does not serve as a base class. This returns true if the number taken in and the colony number match. This is used when ants want to check if they are on their anthill.

**Class TriggerableActor : Public Actor –**

TriggerableActor: Constructs triggerable actor with the specified arguments.

bool isDangerous: I chose to define this as non-virtual because all triggerable actors are dangerous.

virtual ~TriggerableActor: Since TriggerableActor is a base class, it needs a virtual destructor.

**Class Poison : Public TriggerableActor –**

Poison: Constructs poison with the specified arguments.

void doSomething: I chose to define this as non-virtual because Poison does not serve as a base class. This asks the world to poison everything in the objects square.

**Class Pool : Public TriggerableActor –**

Pool: Constructs pool with the specified arguments.

void doSomething: I chose to define this as non-virtual because Pool does not serve as a base class. This asks the world to stun everything in the objects square.

**Class Insect : Public energyHolder –**

Insect: Constructs insect with the specified arguments. Insects have an associated call count to prevent them from being asked to doSomething twice in a tick.

void stun: I chose to define this as non-virtual because all Insects get stunned in the same way, they increase their sleepTurns by 2 if not previously stunned.

virtual void poison: I chose to define this as virtual because one insect has a different way of getting poisoned. This sets the default behavior to taking 150 points of damage and producing food if that results in death.

int sleepTurns: I chose to define this as non-virtual because all Insects report their sleepTurns in the same way, by returning how many turns they will be asleep or stunned

int callCount: I chose to define this as non-virtual because all Insects return their callCount the same way, by returning the tick they were last called on.

bool isStunned: I chose to define this as non-virtual because all Insects return their stun state in the same way, by returning whether they have been previously stunned on their current square or not.

void changeSleepTurnsBy: I chose to define this as non-virtual because all Insects change their sleep turns in the same way, by adding the number that the function takes in and adding it to their sleep turns.

void changeCallCountBy: I chose to define this as non-virtual because all Insects change their call count in the same way, by adding the number that the function takes in and adding it to their call count.

void switchStunState: I chose to define this as non-virtual because all Insects switch their stun state in the same way, by setting it equal to what the function takes in as an argument.

virtual void bittenFor: I chose to define this as virtual because one insect has a different way of getting bitten. This sets the default behavior to reducing health by the specified number, and if that results in death, adding food to the world.

void getForwardCoords: I chose to define this as non-virtual because all Insects get the coordinates of the square in front them in the same way. This function checks the direction that the Insect is facing and from that gets the coordinates of the square in front of the insect. This function is needed for the moveForward function.

bool moveForward: I chose to define this as non-virtual because all Insects move forward in the same way. This function gets the coordinates of the square in front of the insect, checks if the insect can move to the square, and if so will move the insect and return true. If the insect does not move, it returns false. This function is needed for the doSomething function.

virtual void doSomething: I chose to define this function as virtual because, although no derived classes implement a different doSomething, doSomething relies on a pure virtual private function that is different for each derived class.

bool isInsect: I chose to define this function as non-virtual because all insects are insects. This function is used in StudentWorld::move to make sure insects do something before everything else.

virtual ~Insect: Since Insect is a base class, it needs a virtual destructor.

**Class Ant : Public Insect –**

Ant: Construct an ant with the specified conditions. Ants have an associated compiler. Ants must have their call count updated correctly on creation.

bool isDangerous: I chose to define this function as non-virtual because Ant does not serve as a base class. If the number passed in does not match the colony number, this returns true.

bool isEnemy: I chose to define this function as non-virtual because Ant does not serve as a base class. If the number passed in does not match the colony number, this returns true.

void bittenFor: I chose to define this function as non-virtual because Ant does not serve as a base class. This function calls Insects bittenFor and appends a little code that lets the ant keep track of whether or not it was bitten.

**Class Grasshopper: Public Insect –**

Grasshopper: Construct a grasshopper with the specified conditions.

bool isDangerous: I chose to define this as non-virtual because all grasshoppers are dangerous.

bool isEnemy: I chose to define this as non-virtual because all grasshoppers are enemies.

**Class babyGrasshopper: Public Grasshopper –**

babyGrasshopper: Constructs a baby grasshopper with the specified conditions.

**Class adultGrasshopper: Public Grasshopper –**

adultGrasshopper: Constructs an adult grasshopper with the specified conditions. Adult grasshoppers need to have their call count updated correctly on creation.

void poison: I chose to define this as non-virtual because adultGrasshopper does not serve as a base class. Adult grasshoppers cannot be poisoned.

void bittenFor: I chose to define this as non-virtual because adultGrasshopper does not serve as a base class. Adult grasshoppers can retaliate when bitten, so this calls the insect bite function and adds a little to the end.

**Unfinished Functionality/Known Bugs:**

Everything should be functional, and there are no known bugs.

**Design Decisions and Assumptions:**

* If every colony creates an Ant on the same tick, it is counted as though colony 0 created the Ant before colony 1, which created the Ant before colony 2, etc.
* If more than 4 .bug files are specified in the command line arguments, the simulation runs as if there were none (the example executable just crashes).
* The spec and the example executable have different ways of displaying simulation text. I followed the example executables’ style, displaying ticks remaining instead of ticks elapsed and displaying “ants” after the number of ants each colony has created.
* Vectors of actors are visited starting from (0,0) and going to (0,63) before going to the next column.
* Vectors of actors are traversed in reverse order (so erasing will not mess up the positions of the things not yet visited)
* For a grasshopper jumping, two random numbers from 0 to 10 are chosen. If the conditions for a jump are not met, two new random numbers are chosen. It is assumed that this “rejection sampling” is uniformly distributed.
* Adult grasshoppers will try to jump to an empty square 100 times before giving up.
* Insects move first, then everything else does something. Otherwise, grasshoppers will not always be poisoned for the same amount of turns, or they could sometimes skip entirely over pools of water.

**Testing Each Class:**

**Note:** I ran tests on the behaviors of classes that did not serve as base classes, which in effect also tests those classes that do serve as base classes. As such, only classes that do not serve as base classes are listed below.

Class Pebble:

Pebbles only have to block other actors’ movement. This functionality is tested in almost every simulation and test ran. No insect ever moved onto the same square as a pebble.

Class Food:

Food can have health added or removed. I tested this functionality in two separate test cases. In the first, I had ants die on top of food and wrote the food’s health to cerr. It was increased as expected. Second I had ants pick up food and once again wrote the food’s health to cerr. It decreased in the amounts expected. I repeated this test with grasshoppers to test their eating capabilities. I also had ants drop food onto their anthill and had the anthill eat the dropped food, further testing the food object, ant object, and anthill object.

Class Poison:

Poison damages insects every turn an insect is on them. This functionality was tested in two ways. First, the full simulation was run and one grasshopper was surrounded by poison. It died in the correct number of turns in one of the expected spots for it to die. Second, a small arena was created where ants would kill themselves on poison. They died in the expected number of turns.

Class Pool:

Pools stop actors from doing anything for 2 extra turns. This functionality was tested by creating a small arena with pools and seeing what happens when different insects land on them. Grasshoppers stay for 4 ticks. Ants stay for 2 ticks. Afterwards it was checked if they could move after being stunned, but not moving from the pool. Ants were able to rotate, as were grasshoppers.

Class babyGrasshopper:

Baby grasshopper move around randomly, eat, rest, and eventually possibly turn into adult grasshoppers. I tested this functionality by creating a small arena with a baby grasshopper and some food. I wrote to cerr the grasshopper’s health every tick and examined its movement. Eventually it turned into an adult grasshopper, at the health expected. The grasshopper also died correctly in the presence of no food. The baby grasshopper ability to get bitten was also tested using ants in a small arena again (simultaneously testing ants’ biting capabilities).

Class adultGrasshopper:

Adult grasshoppers act almost exactly like baby grasshoppers except they can bite and retaliate. To test their retaliation capabilities, I made it so that the adult grasshopper could not bite and had an ant go bite it. Eventually, the grasshopper bit back, which could be seen through the ant’s response. I then tested the grasshopper’s ability to bite by modifying the chance for it to bite and having an ant walk to it. The grasshopper printed to cerr that it bit the ant. The ant’s health also changed by the correct amount.

Class Pheromone:

Pheromones get placed by ants, occasionally get refreshed, and eventually die. This functionality was tested by creating a small arena and having ants place pheromone upon sensing poison (also testing ant pheromone placing and danger detecting capabilities). Whenever an ant added pheromone, the pheromone health was output to cerr. This confirmed the health cap for pheromones, and that no new pheromone objects were made. As the simulation continued, the pheromone died, confirming that the health was going down as ticks passed. This test was repeated with multiple ants placing their own colony’s pheromones, with similar results.

Class Anthill:

Anthills eat food and produce ants. I tested this functionality by creating a small arena with just food and the anthill. The ants picked up the food and dropped it off at the anthill, which then ate the food and produced more ants. The anthill also tells the world to update its score, so I could see as the ants were being made. I wrote the amount the anthill ate to cerr to be sure it was working properly as well.

Class Ant:

Ants read instructions from a separate file through the use of a compiler. Each possible action and condition needed to be tested. It was assumed that the behavior ants and other insects held in common was tested when examining the other insects (e.g. waiting out stun turns).

To test each condition and action, small arenas were created, like one wide corridors. For example, to test that ants could detect pheromones and danger, I made a corridor with poison at the end. One ant emitted pheromone before the poison. The other ants were to stop in front of the pheromone. To test that ants could detect food, anthills, and whether they were hungry or not, I ran the USCAnt.bug file in a small arena with just food and the anthill and wrote some things to cerr to makes sure everything was working as intended, confirming ant carrying capacity also. Some of the tests I ran for ants are described above, when testing other classes.

Class StudentWorld:

The majority of StudentWorld is tested in the process of testing the other classes. Poison cannot poison without asking the world to do so, Pool cannot stun, Insects cannot bite, Pebbles cannot block, so on, so forth. The functions move(), init(), and cleanUp() are also tested every time a simulation is ran (however, I did run tests on failed compilations). As such, I mainly ran tests on the code that determined the winner. One test was done by adding several food object near one anthill, so that colony would win. Another was done by repeatedly running the simulation to make sure that the first colony to get to the winning number won. I tested the code that created the display text with different numbers of anthills active as well.