1.

**Actor Class:**

The actor is an abstract class since it has a pure virtual function, **doSomething()**, I made this function pure virtual, because actors generally had different tasks/ order of tasks to complete in their doSomething() function. This class has a virtual destructor since it is a base class, so its derived classes need to be destructed as well. The actor class also had a function that returned a pointer to StudentWorld called **getWorld()**, this function was necessary because it allowed me to access members in the student world class. The getWorld() function was not declared virtual, because it requires the same implementation for all the classes. I also included **isDead()** and **setDead()** member functions in order to access and modify the private member variable m\_dead in the student world class. My **converDirToCoord()**, take in the direction and position of the actor and calculated the position of one step in the forward direction. This function was used when an actor was called to move. In addition I also had a **setDidSomething()**, a **resetDidSomething()**, and a **checkDidSomething()** member function to modify and access the boolean m\_didSomething outside the Actor class, in the StudentWorld class.

No pseudo code needed, because all functions are trivial.

**EnergyHolder Class:**

The EnergyHolder class is created as an abstract base class for all objects that have hitpoints. This class has a virtual destructor, because it is a base class, and a pure virtual doSomething() function. The class also has the public member functions **decHitPoints()**, **numHitPopints()**, and **addHitPts()**, these member functions are used by derived classes and by the studentworld class to adjust and check the hitpoints when needed.

No pseudo code needed, because all functions are trivial.

**Insect Class:**

The insect class is a derived class from the EnergyHolder class, since all insects have hitpoints and essential store energy. The insect class is also a abstract base class, as doSomething() is defined as a pure virtual function. The insect class contains my move function for insects, which moves the insect pointer to the specified location if it is allowed to go there. The insect class has helper functions such as **setRandDirection()** which just chooses a random direction and sets the insect’s direction equal to the randomly generated one, which is called in both the ant’s and grasshopper’s doSomething(). I also have member functions that allow access to modify and check the private member variables, m\_distance, m\_canBeStunned, m\_ableMove, m\_stunned, and stunned which simply set the value of one of the variables or return the value of one of the variables.

Pseudo code:

bool Insect::move()

{

change direction and current coordinates to future coordinates

call function in student world to check if move is possible

moves pointer if it is possible

moves display

sets bool values for poisoned, stunned, and didSomething

return true

else

set distance 0 and return false

}

All other functions are trivial.

**GrassHopper Class:**

The GrassHopper class is a derived class of insect and a abstract base class of BabyGrasshopper and AdultGrasshopper. This class has a virtual destructor since it is a base class and its doSomething() function is pure virtual since Baby and Adult Grasshopper have different implementations of the functions. The only other function in Grasshopper is **setDistanceandDirection()** which sets a random distance and a random direction. This is called whenever a grasshopper’s distance is 0.

No pseudo code needed, because all functions are trivial.

**Baby and Adult Grasshopper Classes:**

The baby and adult grasshopper classes are a derived class of GrassHopper and only contain a constructor and empty destructor, and a doSomething function. These classes did not need any extra data members or functions, because they were already created in the insect and grassHopper class.

void BabyGrasshopper::doSomething()

{

check if already done something return if has

set did something, dec hit points

if hitpoints <=0 setDead and return

if stunned or sleeping decrement respective variable and return

if hitpoints are high enough turn to adult grasshopper and set baby dead and return

if food on spot eat and return

if distance = 0 set a new distance and direction and move one step if possible

reset able move

}

void AdultGrasshopper::doSomething()

{

heck if already done something return if has

set did something, dec hit points

if hitpoints <=0 setDead and return

if sleeping dec sleep count and return

//after each of the following set sleep ticks to 2

choose if want to bite try to bite another insect and return

choose if want to jump if so try to jump return

eat if food is available return

move one step in the current direction if can return

}

**Pebble Class:**

The pebble class like the adult and baby grasshopper classes only has a constructor a destructor and a doSomething function. The pebble class is even simpler, because the body of all three of these functions is empty (other than the initialization of the Actor base class).

No pseudo code needed, because all functions are trivial.

**WaterPool Class:**

The waterpool class is a derived class of the actor base class and only has a constructor destructor, and a doSomething() function.

void WaterPool::doSomething()

{

check if already stunned

set did something

iterate through all stunnable insects and stun if not already stunned

}

**Poison Class:**

The Poison class is a derived class of Actor, it contains only three functions the constructor, the destructor, and its doSomething function which decreases the hit points of the insects (which can be poisoned) which reside on the spot.

void Poison::doSomething()

{

iterate through all poisonable insects and poison if not already stunned

}

**Food Class:**

Food is a derived class of EnergyHolder since it has hitpoints (amount of units of food left on the spot). Since food does not do anything and instead gets acted upon by objects of other classes. The only functions are the constructor, destructor, and the doSomething() function with and empty body.

No pseudo code needed, because all functions are trivial.

**Ant Class:**

The Ant class is a derived class of insect, the ant Class has many public member functions in order for it to interpret the instructions given to it by the competitors. Like the other classes the Ant class has a constructor, destructor, and a doSomething function. In addition, it also has an **executeUpto10Commands()** function that executes commands given to the ant by the competitor either until the state of the field is changed or 10 commands from the .bug file are executed. In addition, I created another function called executeIfCorrect(), which identifies which instruction to set my instruction counter to by evaluating the condition provided by the passed in command. I also have trivial functions such as getColNum(), EatFood(), setBitten(), setUnbitten(), rotateClockwise(), and rotateCounterClockwise() which modify private data member or give access to it.

void Ant::doSomething()

{

heck if already done something return if has

set did something, dec hit points

if hitpoints <=0 setDead and return

if stunned decrement respective variable and return

call execute up to 10 commands

if this returns false there was a problem set ant to dead

set commands execute to 0

}

bool Ant::executeUpto10Commands()

{

check to make sure that command is valid

if not return false

switch statements that call respective functions for respective commands

return true

}

bool Ant::executeCorrectIf(Compiler::Command command)

{

switch statements that check if the condition is true

if true set instruction to operand 2

else set instruction to next instruction

return

}

**Anthill Class:**

The anthill class is a derived class of EnergyHolder, because of its use of hit points. It contains a constructor, destructor, doSomething() function and a function that returns a pointer to its compiler object to pass into ants when it creates them.

void Anthill::doSomething()

{

dec hit points set did something check if dead if dead return

check for food if there is intake and return

if hitpoints are greater that 2000 create a new ant and set hitpts

inc count of ants made in each colony

check who current winner/leader is

}

**Pheromone Class:**

The Pheromone class is a derived class of EnergyHolder. because it contains hitpoints. The Pheromone class has a constructor, destructor, doSomething() function, as well as one trivial function **getColNum()** which returns the colony number of the pheromone.

void Pheromone::doSometihng()

{

dec hit points if hitpoints is <=0 set dead

}

**StudentWorld Class:**

In my student world class I modify the actual field in which the pointers to the actors lie. This required many member functions since, the the actors called functions from student world to execute moves. Some trivial functions included in the StudentWorld class are **updateTickCount()**, **incNumAntsinCol()**, **getColonyID()**, **getPherID()**, and **AntWasBitten()** all of these functions either modify a private member variable by incrementing/decrementing or give access to the value of data member. The **init** function is virtual and it places the objects in the correct spots in the private member array. In addition it also passes compiler objects into each anthill for the ants to use. The **move()** function calls do something on each member in the array and deletes all dead actors at the end of every tick. The **cleanUp** function, called at the end of the simulation deallocates from memory all the objects created. The functions I chose to implement are called in init and move as well as functions in the Actor class. I put the functions that had to do with formatting (**setDisplayText()**, **Format()**, **setAstForLeader()**) inside the student world class, because they would be called every tick, so it was more convenient to have it inside the StudentWorld class, so it could have access to its private member variables. The functions **blocksInsect()**, **checkMove()**, **jump()**, **insectsAtPos()**, **bite()**, **isFoodOnSpot()**, **addFoodOnSpot()**, **biteBack()**, **stun(), antBite()** all modify the placement of the actor pointers in my private data member, so it made sense to put them in student world so it the function s would be able to easily switch the pointers from grid point to grid point. The same logic goes for the add and locate functions like **addAdultGHop()** and **enemyOfAnt()** which also need access to the field array and its vectors to place or locate actors. In addition to these member functions I also had a template function that given a coordinate point and a pointer to a specific type of actor, whether that type of actor was on that spot or not.

Pseudo Code:

init()

{

load field file

iterate through each point in the grid and dynamically allocate objects in right location

create compiler objects

compile the objects

create new anthill with objects passed in

create an array with the names of all the colonies for later use

return continue game

}

move()

{

update tick count

ask each actor to do something

remove dead objects

reset all didsomething member variables

display the text

if ticks not over continue

else find winning ant name and display if no winner return no winner

}

SetDisplayText

{

creates ints

creates string calls Format

displays formatted string

}

Format

{

created ostringstream

call setAstForLeader

added elements necessary based on number of ant colonies

return

}

setAstForLeader

{

takes winningCol number

adds asterisk in correct place based on it using switch

}

getWinningAntNumber

{

loads array of num ants in each colony

compares current winning against others (if no winning so far start from element 0)

only changes if strictly greater than

if winner ants = 5 -> return -1 since didnt make any

else return winner

}

cleanUp()

{

iterate through all the vectors and delete the dynamically allocated objects

push nullptr into each vector

}

removeDeadObjects()

{

iterate through all vectors if something is dead

check if it is an insect

if yes add food to spot and delete insect

else delete actor

}

resetAllDidSomething()

{

goes to each actor and sets m\_didSomething to false

}

blocksInsect

{

goes to specific vector at x, y

dynamic casts each pointer in vector to pebble

if not equal to nullptr return false //pebble in the way

}

checkMove

{

calls blockInsect

returns if pebble blocking

push pointer passed in to the new destination

iterate through old vector to find place of obj moved

assign back of vector to that position

pop the back of the vector

return true

}

isFoodOnSpot

{

iterate through vector

dynamic cast to food pointer

if there take food/hitpoints

return the amount of food taken

if no food return -1

}

addFoodOnSpot

{

iterates through vector if food is there

adds food to that food actor

else dynamically allocates a new one at that point

}

stun

{

iterate through vector

if there is an insect and it can be stunned and has not been stunned yet

stun it

set it to stunned

}

poisonn

{

iterate through vector

if there is an insect and it can be poisoned and has not been poisoned yet

dec the insects hit points

if dead set it to dead

}

addNewAnt

{

dynamically allocate new ant with passed in colony number

}

addAdultGHop

{

dynamically allocate adult grasshopper

}

jump

{

get random radius and theta

rcos(theta), rsin(theta) to ints

if valid coordinates

return

else recursion

}

bite

{

a number, n, is passed in

the nth insect is the one who needs to be bitten

find the nth insect and bite

see if it wants to biteback

}

emitPheromone

{

see if pheromeone exists on the spot

if so add pheromone up to 768 to pheromone

else

add new pheromone object with correct hitpoints

}

AntBite

{

get how many enemies there are

chose random nth enemy

find it in array

bite

if dead set dead

else

see if it wants to bite back

}

biteBack

{

if the insect is not adult grasshopper return

else randInt for 50% chance

if wants get insects at vector

picks random one to bite

return bite

}

enemyOfAnt

{

iterates and checks if actor is insect if it is an insect

checks if ant with different colony

if so increases count of enemies

}

Template function

{

iterate through vector

dynamic cast to type specified

if not equal to nullptr return true

else false

}

2. I believe everything works correctly.

3. It was not specified what to do when an anthill died, so I decided that if al anthill died so would the ants, because its compiler object is destructed. In addition the main objective of the game is to make the most ants, so if the anthill dies that means that no more ants can be made, so what happens to the ants does not matter very much.

4.

**Ant Class:**

As I was writing my code for Ant, I was periodically running it to see if I was getting any build errors. Once I finished writing the program I first just observed the ant’s behavior with USCAnt and saw what it did. I examined the ants more closely by using the pause functionality. I made sure the ants were moving every tick and stopping at the pool of water at the right amount of time. Since the USCAnt did not implement all of the functionalities of the ant, I created my own .bug files that included more if statements and pheromones. This way I could see that the pheromones worked. Once I was satisfied with how it worked as a whole I started looking more closely at the details, and created .bug files with only one or two functions implemented to see that the ant was completing the same task correctly over and over again. For example, I tested out the rotate clockwise and move functions, by only putting those two things in my .bug file and making sure it was actually turning and moving in the correct direction.

**Pebble Class:**

I tested my pebble class by making sure there were pebbles around the border, and that no pebbles disappeared. I also tested it by making sure that the pebbles were in the right spot by comparing their placement to the field.txt file that was provided.

**Food Class:**

I tested the food class mostly by using cout statements and by watching. I used cout statements to see how many hitpoints the food had and made sure that the bugs were taking it off in the increments specified. The pause function worked very well in this case, because I could find a insect that moved onto food and see the change in the food’s hitpoints.

**Pheromone Class:**

I tested the pheromone class by creating my own .bug file where the ant was told to drop pheromones often. I looked to make sure that the pheromones were disappearing at about the right time. In addition I made sure that each ant was emitting the correct color of pheromone.

**Anthill Class:**

I tested the anthill, by first making sure that it birthed only 5 ants in the first 5 ticks, I also used cout statements to see how the hitpoints were decremented and to make sure they were decremented correctly. Also I made sure to check that no error occurred when an anthill was destructed, because it ran out it hitpoints. When this happens all the ants are set to dead, because the compiler object does not exist anymore, so the ants have no instructions.

**Pool of Water Class:**

I tested my pool of water class after I created the baby grasshoppers, so I could see if the pool of water was actually causing the grasshoppers to stay still for 2 extra turns. I did this by pausing the program and running through it tick by tick. This helped my see that the grasshoppers were staying at the same spot for exactly 4 ticks. I then reexamined the pool of water behavior when I made the ant class making sure it was stuck for 2 ticks. One of the problems I encountered while trying to see the insects were stunned for the correct amount of time is that sometimes ants are not stunned but don’t perform any external action either. To fix this case, I used cout statements to see how many ticks left they had to be stunned.

**Poison Class:**

I tested my poison class by using cout statements and by observing that insects do die when on the poison. I tested this more in depth by using cout statements and printing out the hitpoint of the insect before and after poison is called. I made sure that baby grasshopper and ant were getting the correct negative amount and that adult grasshopper does not lose any hit points.

**Baby Grasshopper Class:**

I tested the baby grasshopper incrementally. I first tested if the grasshopper would move then when I implemented more I would make sure it would sleep and get stunned on the correct ticks. After I created the adult grasshopper class I made sure that baby grasshoppers truned into adults by using cout statements, to print out the hitpoints before and after the change.

I mostly tested the baby grasshopper by pausing and observing it and making sure that they were stopping at pools of water and sleeping two out of three ticks. I also made sure they turned into food when they die by not letting the grasshopper eat/gain hit points.

**Adult Grasshopper Class:**

Testing the adult grasshopper class was more difficult than testing the baby grasshopper class, because there were more functionalities that were implemented during a smaller portion of the ticks. I tested the adult grasshopper bite function by making it bite an insect every time it was on the same spot as one. This allowed me to see the decrease in hit points in a more dramatic sense. To test the jump function I commented out the bite function and made the grasshopper jump every time and made sure that it was around 10 squares of its original position. I also made sure that grasshoppers only move every 3 turns by using the pause functionality of the program.