Class StudentWorld

\*\*None of these are pure virtual so that not every new class has to declare a function. The default function serves as a default/base case.

virtual int init();

Create a field, compile it, and return an error if not able to compile.

Make an array of pointers compilers

Get a vector of the Ant Programs

For loop running through however many ant programs there are:

Create a compiler for each program and point to it via the array

Have a separate array telling us which anthill is usable.

For loop for each row and column:

If the item is a rock, babygrasshopper, food, poison, water, or anthill, dynamically allocate that item.

virtual int move();

For loop for each square on the grid:

Set the moved state to false

Increase the tick count.

For loop for each square:

Run through the lists of pointers to actors and make it dosomething.

Set the items to moved.

If necessary, delete a pointer and point to the object from elsewhere.

For loop for each square:

Run through the list via an iterator and delete all dead actors.

Display the updated stats.

Continue the game until 2000 ticks.

If there is a winner, declare the winner.

virtual void cleanUp();

For loop for each square on the grid:

Run through the list and delete all Actors.

bool isThisBlocked(int x, int y);

This uses an iterator to run through the Actors on the square in order to determine if there was any Actor capable of blocking the motion of another Actor onto the square (such as a pebble). I chose it in this class because it will be able to access other actors stored in the list.

void addFoodHere(int x, int y, int amount);

This runs through Actors on the square to see if there is already a Food or if it has to create a Food to add to. Reason: Student World has access to all Actors on the square.

int howMuchFood(int x, int y);

This runs through Actors on the square to see if there if is already Food and how much. Reason: Student World has access to all Actors on the square.

void addPheromoneHere(int x, int y, Actor\* me);

This runs through Actors on the square to see if there is already a Pheromone from the same ant or if it needs to be created in order to add to. Reason: Student World has access to all Actors on the square.

bool myPheromoneHere(int x, int y, Actor\* me);

This runs through Actors on the square to see if there is a Pheromone from itself -> used for ants. Reason: Student World has access to all Actors on the square.

void addAntHere(int x, int y, int colID, Compiler\* compilerCode, AntHill\* anthill);

This adds an ant onto the square and contains all the items for the construction. Created in this class because the StudentWorld holds all the actors.

void addAdultGrassHopperHere(int x, int y);

This creates an adult grasshopper where a baby one has died. It is fitting because of the fact this class holds all of the actors.

bool biteSomeoneHere(int x, int y, Actor\* me);

This runs through all the actors on the square and finds bitable ones to randomize. It also damages based off of the insect type. When biting, it will call itself recursively if the bitten insect can bite back. This class can run through all the actors on the square.

void stunHere(int x, int y);

This runs through all the actors here and increases the stun on the stunnable ones (insects). It can be called by an actor and in StudentWorld is allowed to access other actors.

void poisonHere(int x, int y);

This runs through all the actors here and finds the ones that are able to be poisoned and then poisons them.

bool dangerHere(int x, int y, Actor\* me);

This runs through actors (only this class can do) and determines whether there are dangerous actors here (enemy ants, grasshoppers, water, etc.)

bool enemyHere(int x, int y, Actor\* me);

This runs through actors (only class can do this) and determines whether there are enemy insects on the square.

Class Actor

virtual void doSomething() {};

This remains virtual instead of pure virtual because though actor needs one, pebbles do not differ. If some other object like a boulder is added in, they would not need to redefine this because it is not pure virtual.

virtual bool blocksAnts();

All actors are able to either block ants or not so this is in the base most class. Because they may or may not, it is virtual so that they can be defined for the specific class.

virtual bool isFood();

This is used in adding and counting food in student world and thus is added for all actors as the world only has pointers to actors. It is virtual because not everything is food.

virtual bool isPheromone() { return false; };

This is used in determining pheromone in student world and thus is added for all actors as the world only has pointers to actors. It is virtual because not everything is a pheromone.

virtual bool isDangerous() { return false; };

This is used in determining danger in student world and thus is added for all actors as the world only has pointers to actors. It is virtual because not everything is hazardous.

void setAliveDead(bool value);

This sets the actor to alive or not. It edits a private member of actor and thus needs to be in that class.

Bool isAlive();

This asks if the Actor is alive or not in order to determine whether to delete them or not via student world.

void setMoved(bool value);

This sets the Actor to moved/not moved so that it does not move multiple times in one tick.

bool moved();

This tells the StudentWorld whether an actor was already moved.

virtual void adjustEnergy(int x) {};

This adjusts the energy of an Actor but is needed in the base class because StudentWorld has pointers to Actors, not specifics. It’s virtual so that those that have energy can actually do something.

virtual int checkEnergy() { return -1; };

This checks the energy of an Actor but is needed in the base class because StudentWorld has pointers to Actors, not specifics. It’s virtual so that those that have energy can actually do something.

virtual bool ableToBeStunned() { return false; };

virtual int stunnedAmount() { return -1; };

virtual void adjustStunned(int x) {};

These check the stun amount/ability to be stun of an Actor but is needed in the base class because StudentWorld has pointers to Actors, not specifics. It’s virtual so that every stunnable object can act.

virtual bool ableToBePoisoned() { return false; };

This checks if something is able to be poisoned or not and is virtual because not all actors have the same response.

virtual bool ableToBeBit() { return false; };

virtual bool willBiteBack() { return false; };

virtual bool amIBit() { return false; };

virtual void setBitAlreadyState(bool value) {};

These functions deal with biting which the studentworld does and thus it needs to be in the base class so that the student world may call upon it. The virtual portion will serve to allow difference responses for the different classes.

virtual int colony() { return -1; };

This returns a colony number but is only applicable to ants. However since the student world class needs to use it, it must be virtual and in the base class.

StudentWorld\* myWorld();

This returns the studentWorld so that each actor may call it for an action.

class Pebble :

virtual bool blocksAnts();

This returns true because pebbles are the only object (now) that block ants.

Class Water

virtual bool isDangerous() { return true; };

This modifies the default actor’s case of whether it is dangerous or not.

void stun();

This calls student world to stun the objects on the same square.

Class Poison

virtual bool isDangerous() { return true; };

This modifies the default actor’s case of whether it is dangerous or not.

void poison();

This calls student world to poison all objects on the same square depending on what they are.

Class EnergyHolder

void adjustEnergy(int x);

This modifies the private variable to adjust the energy. It is not virtual because it will always be able to adjust hit points or health points or food points as the same type.

int checkEnergy();

This checks how much energy the Actor has left.

Class Food

virtual bool isFood();

This returns true for food and is called by studentworld and thus is virtual.

Class Pheromone

virtual void doSomething();

The Pheromone decreases 1 in health everytime.

virtual bool isPheromone() { return true; };

This overrides the studentworld call.

virtual int colony();

This reads out the private data member that holds the colony number so it can be traced back to which ant colony it belongs to.

Class AntHill

virtual void doSomething();

The anthill decreases in energy by 1 every turn.

Checks if there is food

If so -> add food to own energy points

If energy is greater than 2000, make a new ant.

class Insect

bool adjustmentsAndSleep();

All the insects decrease their health by one and then decrease their stunned state by one as their first move.

virtual int stunnedAmount();

virtual void adjustStunned(int x);

All insects are able to get stunned and thus are able to adjust and read out their stunned state.

bool ableToBeBit() { return true; };

Insects are able to be bit and thus override the actor function.

virtual bool isDangerous() { return true; };

Insects are dangerous to one another and thus override this actor function.

class GrassHopper : public Insect

bool eatFoodHere();

This calls the student world and eats 200 units of food or whatever is available up to 200 units of food.

void moveInADirection();

All grasshoppers are able to walk in the same sort of pattern.

Get the direction that the grasshopper is facing

Increase 1 unit in that direction

If they can move there, move to that location

Else

Set remaining distance to 0 and stun itself for 2 ticks.

int remainingDistance();

Reads out the remaining distance to travel in current direction.

void setRemainingDistance(int x);

Set remaining distance to x.

class BabyGrassHopper

virtual void doSomething();

Calls the function to adjust health by 1 and sleep if necessary

Check if energy is 1600 or more

If so die, add 100 food to the square and respawn as an adult.

Eat food on square if present

If accomplished return.

If no more distance to walk, set a new distance and direction.

Move in stated direction for 1 distance.

virtual bool ableToBeStunned() { return true; };

virtual bool ableToBePoisoned() { return true; };

These are true for the baby grasshopper but not the adult and thus is in this

class.

Class AdultGrassHopper

virtual void doSomething();

Calls the function to adjust health by 1 and sleep if necessary

1/3 chance of biting someone -> determine with random integers

If bit someone, stun itself for two turns and return.

1/10 chance of jumping -> determine with random integers

If jumping to a random location in the circle radius 10 is successful

Move to that location and stun itself for two turns and return.

If there is food on the square, eat it and return.

Otherwise, move in a direction as baby grasshoppers did.

If it is moved, the bit on this square and blocked on this square read out false.

virtual bool willBiteBack() { return true; };

Adult grasshoppers are the only insect that will bite back as of now.

class Ant

virtual void doSomething();

Calls the function to adjust health by 1 and sleep if necessary

If the interpreter can’t read the move that the ant functions with

Set the ant to dead.

Otherwise, move according to the interpreter.

virtual bool amIBit();

Edits the private data member and returns whether it was bit or not.

virtual void setBitAlreadyState(bool value);

Set the state of bit on the current square.

virtual int colony();

This reads out the ant’s colony so that we can identify it.

My program seems to have all of the functions implemented but it crashes randomly for unknown reasons. However, I think I might have fixed this. I am not sure.

For the adult grasshopper, whenever it went to step 13, it said “set the number of ticks to sleep to 2” so I set the sleep/stun ticks to 2 because it was unclear whether to add it or not.

To test the pebble class, I checked whether it appeared or not.

To test the water and poison class, I looked at its effect on the ants and grasshoppers and how they moved. I used the “f” key to freeze the screen and step through the instances.

To test the baby grasshoppers, I ran through the program and traced it with the debugger. Additionally, I froze the screen to check their movement. Tracing it with the debugger allowed me to see what sleep ticks they had and if they were stunned or not by the water and poison.

To test adult grasshoppers, I set it so that it could only hop or had to bit other members on the same square of the grasshopper so that it was forced to act. I also tested out the basic motions at the same time as the baby grasshopper as the code was shared and very similar.

To test out ants, I tested out each command separately to see if it worked. Additionally, I used the cerr function to read out text to see if the function was actually running through or simply breaking out of the switch statement.

For all of the classes, for virtual functions that they overrode, I made them cerr something out so that I could double check whether they were actually overriding the virtual function for the base Actor class.