REPORT

PROJECT 3

DESCRIPTION OF EACH PUBLIC FUNCTION

ACTOR

Constructor for Actor

Passes id, coordinates, direction and depth into the graph object, sets the world, sets the actor as not dead and not sleeping.

Constructor cannot be made virtual.

Actor::doSomething

This has a different implementation for every single derived class, so this was made pure virtual.

Actor::Health()

This simply returns the current health of the actor.

This works exactly the same for every object, so not needed to be virtual.

Actor::isDead

This simply returns whether the actor is dead or not.

This works exactly the same for every object, so not needed to be virtual.

Actor::setDead

This sets the actor to dead.

This works exactly the same for every object, so not needed to be virtual.

Actor::setHealth

This sets the health of the actor to the new passed value.

This works exactly the same for every object, so not needed to be virtual.

Actor::sleeping and Insect::sleeping

This increments the sleeping counter by the passed value and returns the new value. In case user needs to check the current value, they can do it by passing 0.

The actor function is virtual, since it has to be utilized by Insect class differently.

Actor::biting and Insect::biting

This sets whether the actor can bite (positive biting power) or not (0 biting power).

The actor function is virtual, since it has to be utilized by Insect class differently.

Actor::canBeBitten and Insect::canBeBitten

This sets whether the actor can be bitten or not. This is mainly used for debugging purposes.

This works exactly the same for every object, so not needed to be virtual.

The actor function is virtual, since it has to be utilized by Insect class differently.

Actor::id

This simply returns the unique id of every actor. Although some actors use it in different ways, different implementation of such a simple thing is not required. Hence, this was not made virtual or pure virtual.

Actor::changeId

This is used to change the id of the actor.

This works exactly the same for every object, so not needed to be virtual.

Actor::world

This returns the home world of the actor.

This works exactly the same for every object, so not needed to be virtual.

Actor::dirResolver and Insect::dirResolver

On the basis of which direction is passed, it changes the coordinates by 1 in a particular direction.

The actor function is virtual, since it has to be utilized by Insect class differently.

Actor::notDeadAnymore

Sets the state of the actor to alive.

This works exactly the same for every object, so not needed to be virtual.

Actor::poolSleeping and Insect::poolSleeping

Simply returns whether an object has been stunned because of a pool of water or not. This is required to make sure a pool of water doesn’t stun the same actor multiple times.

The actor function is virtual, since it has to be utilized by Insect class differently.

Actor::poolSleeping(bool changer) and Insect::poolSleeping(bool changer)

This is used to toggle whether an actor has been stunned by a pool.

The actor function is virtual, since it has to be utilized by Insect class differently.

Actor::randDir and Insect::randDir

This generates a random integer between 1 and 4 inclusive. It then resolves it into a direction based on the integer generated earlier.

The actor function is virtual, since it has to be utilized by Insect class differently.

Pebble::Constructor

Passes the home world, and the coordinates. Sets the correct ID for the graph object.

A constructor cannot be virtual.

Pebble::doSomething

This is a dummy function which does nothing, but has to exist because it was defined as pure virtual in the base class.

Grasshopper::Constructor

Sets the id, direction, home world and coordinates.

A constructor cannot be virtual.

Grasshopper::doSomething

Defined as pure virtual, because both the classes which derive from Grasshopper do different things with it.

Grasshopper::dist

Returns the distance which the grasshopper should be travelling in the direction it is facing.

Not needed to be virtual because both the classes which derive from Grasshopper use it in the same way.

Grasshopper::changeDist

Changes the distance to be travelled to the new passed value. Not needed to be virtual because both the classes which derive from Grasshopper use it in the same way.

Not needed to be virtual because both the classes which derive from Grasshopper use it in the same way.

The functions from this point on don’t need to be virtual or pure virtual at all, since no classes are derived from the classes to which they belong.

BabyGrasshopper::Constructor

Sets the home world, coordinates and appropriate ID, along with initial health and biting power.

Constructors cannot be virtual.

BabyGrasshopper::doSomething

PSEUDOCODE:

Reduce health by 1

If actor is dead

Return

If health is less than or equal to 0

Add food where the actor is standing

Set status of actor to dead

Return

If sleeping counter is greater than 0

Reduce sleeping counter by 1

Return

Set poolSleeping to false

If health is greater than or equal to 1600

Kill the baby grasshopper, add food to the spot

Create an adult grasshopper at this spot

Return

If there is food on the current spot

Attempt to eat food on that spot

Increase health by amount of food that was successfully eaten

Generate a number which is either 1 or 2, if the number is 1

Increase sleeping counter by 2

Return

If distance to be travelled by actor is 0

Make it face a random direction (use randDir)

Set target distance to a random integer between 2 and 10 inclusive

Store the coordinates which are 1 step ahead in the currently facing direction

If there is a pebble on those coordinates

Set target distance to 0

Else

Move the actor to the new coordinates (change in graph object)

Reduce target distance by 1

Increment sleeping counter by 1

BabyGrasshopper::isBitten

This reduces the health of the grasshopper by the biting power, and also checks whether it died in the process or not.

AdultGrasshopper::Constructor

Sets the home world, coordinates, appropriate ID, random starting direction, starting health to 1600 and biting power to 50.

AdultGrasshopper::doSomething

PSEUDOCODE

Reduce health by 1

If grasshopper is dead

Return

If health is less than or equal to 0

Add food to this spot

Set grasshopper to dead

Return

If sleeping counter is greater than 0

Reduce sleeping counter by 1

Return

Set poolSleeping to false

Generate a random integer between 1 and 3, if that number is 1

If there is an “alive” actor on that spot (alive means that actor is Ant or a Grasshopper)

Try to bite it

If bite was successful

Increase sleeping counter by 2

Return

Generate a random integer from 1 to 10, if that integer is 1

Check for empty spot around grasshopper using findSpot()

If spot successfully found

Move to that spot

Return

If there is food where the grasshopper is standing

Attempt to eat that food

Increase health by the amount of food successfully eaten

Generate a random integer from 1 to 2 inclusive and if that number is 1

Increase sleeping counter by 2

Return

If target distance is 0

Make grasshopper face random direction

Set target distance to a random integer from 2 to 10 inclusive

Use dirResolver() to try to move a step forward

If blocked by pebble

Change target distance to 0

Increase sleeping counter by 2

Return

Else

Move to the new spot

Decrement target distance by 1

Increase sleeping counter by 2

Return

AdultGrasshopper::isBitten

Reduce health of grasshopper by bitingPower and check if grasshopper died in the process.

There is a 50% chance that we ask the world to bite whatever there is on that spot with bitingPower of 50. We pass the this keyword as well to make sure the grasshopper doesn’t bite itself.

AdultGrasshopper::findSpot(int &x, int &y, int r)

PSEUDOCODE

Go through every cell in the grid

If distance of that cell from x and y is less than or equal to r

(calculated using distance formula) then:

if there isn’t a pebble on that spot

If that cell isn’t the original cell passed

Set the coordinates to the new coordinates

Return

Food::Constructor

Make the food start out with 6000 hp.

Food::doSomething

Dummy function required here, since doSomething was declared pure virtual in Actor class.

PoolOfWater::Constructor

Passes the home world, coordinates, appropriate ID, direction and depth.

PoolOfWater::doSomething()

Asks world to stun everything stunnable on its current coordinates.

Pheromone::Constructor

Pass the id, home world, coordinates and appropriate colony. Set starting health to 256 and changeID to colony so it can be accessed by world to uniquely identify it.

Pheromone::doSomething

Reduce health by 1 and check if it died in the process.

Poison::Constructor

Passes the home world and the coordinates.

Poison::doSomething

Asks world to poison all posionable things on those coordinates.

Anthill::constructor

Passes the homeworld, coordinates and the colony number. Sets starting health to 8999, and saves the compiler pointer.

Anthill::doSomething

PSEUDOCODE

Reduce health by 1

If health is less than 0

Set status to dead

Return

If there is food where grasshopper is standing

Try to eat that food

Increase health by the amount of food eaten

Return

If health is greater than or equal to 2000

Make a new ant based on the colony number of the anthill

Ask world to put this ant in the data structure

Reduce health by 1500

Ant::Constructor

Passes the ID, passes the home world, coordinates, colony number, and pointer to compiler.

It sets the starting health to 1500, starting food held to 0, rowNumber of compiler to 0, biting power to 15.

Ant::interpreter

PSEUDOCODE

Increase depth count of interpreter by 1

If command is the following, then do something:

Move forward

Try to move forward using tryToMove()

eatFood

Eat the food held, and make sure only 100 food is eaten at a time

dropFood

Ask world to drop food on that spot

Set food currently held to 0

Bite

Ask world to bite something with biting power of 15

pickupFood

If there is food on that spot, attempt to pick up 400 food, or less if more than 1400 food is held by the ant currently

Increase food held by the amount of food successfully picked up

emitPheromone

Ask world to create pheromone

faceRandomDirection

Ask ant to face in a random direction

generateRandomNumber

Convert operand1 string to integer using stringstream

Generate random integer between 0 and limit inclusive and save it in private member

Go\_to

Record that a goto command was given

Convert Operand1 to integer using stringstream

Change row number to operand1

If\_command

Record that if command was given

Convert operand1 and operand2 to integer using stringstream

If operand1 is the following, then do something:

Last\_random\_number\_was\_zero

If random saved number was 0

Set row number to operand2

I\_am\_carrying\_food

If food held is more than 0

Set row number to operand2

I\_am\_hungry

If health is less than or equal to 25

Set row number to operand2

I\_am\_standing\_with\_an\_enemy

If there is an enemy standing there on that spot

Set row number to operand2

I\_am\_standing\_on\_food

If the ant is standing on food

Set row number to operand2

I\_am\_standing\_on\_my\_anthill

If the ant is standing on its anthill

Set row number to operand2

I\_smell\_pheromone\_in\_front\_of\_me

Get the coordinates of the spot in front of ant using dirResolver()

If there is pheromone on those coordinates

Set row number to operand2

I\_smell\_danger\_in\_front\_of\_me

Get the coordinates of the spot in front of ant using dirResolver()

If there is an enemy on those coordinates

Set row number to operand2

I\_was\_bit

If ant was bitten in last tick

Set row number to operand2

I\_was\_blocked\_from\_moving

If ant was blocked from moving in las tick

Set row number to operand2

Ant::isBitten

Reduce health of ant by biting power, and check if it died in the process. If it died, then add 100 food to that spot. Record that the ant was bitten in this tick.

Ant::tryToMove

Get coordinates of the cell a step forward in the direction of the ant using dirResolver() and check if that spot has a pebble or not. If it’s free, then move ant to it, and record whether the ant was blocked or not.

Ant::doSomething

PSEDUOCODE

Reduce health by 1

If ant is dead

Return

If health of ant is less than or equal to 0

Add 100 food to world

Set status of ant to dead

Return

If sleeping counter is greater than 0

Reduce sleeping counter by 1

Return

Set that ant has not been stunned by a pool

If compilation of instructions for this ant was not successful

Return

Create a new empty command

Create an integer interpreter depth counter variable

Create an integer variable to keep track of which command was executed

Start infinite loop

Store instruction in rowNumber line to empty command created previously

If retrieving that instruction didn’t work

Set status of ant to dead

Return

Ask interpreter function to make sense of this instruction

Record that the ant wasn’t bitten in this tick

If go\_to or if\_command were not executed

Increase rowNumber by 1

If a command was executed which required loop to end now

Return

If depth of interpreter has reached or exceeded 10

Return

STUDENTWORLD

StudentWorld::constructor

Clear the data structure

Initialize the ant produced array

StudentWorld::~StudentWorld

Let the clean up function do the work

StudentWorld::init

Initialize tick counter

Initialize compiledCorrectly to false

Get the filenames of the Ant Programs

For every cell in the grid, get contents of the field

Create an object according to the retrieved content

Push it into the data structure

StudentWorld::move

Reduce tick counter

If tick counter is less than or equal to 0, check if anyone is winning and return the appropriate game status after setting winner.

Check who’s winning the game and update data accordingly.

Call do something function for every object in the grid

Check if the stuff moved after “doing something”, and if it did, move it around in the data structure accordingly.

Check if some stuff died, and remove it from the data structure if it did.

StudentWorld::updateDisplayText()

This develops the game text to be displayed at the top of the simulation.

StudentWorld::cleanup()

Delete all the objects

StudentWorld::addFood

Add food of the given amount to the given coordinates. Create new object if required and push it into the data structure.

StudentWorld::createAdultGrasshopper

Creates a new grasshopper and pushes it into the data structure.

StudentWorld::isPebble

Checks if there is a pebble in the given spot.

StudentWorld::isAlive

Check if there is an insect on that coordinate.

StudentWorld::biteSomething

Check if something biteable is there and make sure its not the biter. Bite it if possible.

StudentWorld::isFood

Check if there is food on that spot with more than 0 health.

StudentWorld::eatFood

Attempt to eat food in that spot.

StudentWorld::stunThem

Attempt to stun everything stunnable on that spot

StudentWorld::poisonThem

Attempt to poison everything on that spot

StudentWorld::createAnt

Create a new ant and push it into the data structure.

StudentWorld::createPheromone

Create a new pheromone of the appropriate colony number and push it into the data structure.

StudentWorld::checkCompilation

Return the compilation status of a particular ant’s instructions.

StudentWorld::isEnemy

Check if an enemy actor is present on that spot.

StudentWorld::isMyAnthill

Check if the ant is standing on its own anthill.

StudentWorld::isPheromone

Check if there is a pheromone on the given spot.

FUNCTIONALITY

As far as I know, I finished all the required functionality of the simulation. I have not encountered any bugs which I didn’t fix yet.

DESIGN DECISIONS

I did not encounter anything which wasn’t very clearly defined the specification. Hence, I did not have a situation in which I had to use my own design thoughts instead of looking at the specification.

TESTING OF CLASSES

PEBBLE

The pebbles were being displayed correctly in the simulation, which I confirmed by putting in various pebbles in different places in field.txt.

I checked if the pebbles were correctly recorded in the data structure by using cout to view the contents of the data structure.

FOOD

The food items were being displayed correctly in the simulation, which I confirmed by putting in various food items in different places in field.txt.

I made sure that the food items were holding the correct health and they were being recorded correctly in the data structure by using cout. I also checked that they vanished upon death and appropriate reduction of health by manually killing them after a particular amount of ticks. I checked that the actors which ate the food had their health increased by using cout on their current health variables whenever they ate food.

PHEROMONE

The pheromones were being displayed correctly in the simulation, which I tested by adding the appropriate instructions to USCAnt.bug. I checked that they were being recorded correctly in the data structure by using cout. I also checked their health declination and that multiple pheromones of the same type did not exist on the same spot by using cout for all pheromones and printing their coordinates.

POOL OF WATER

The pools of water were being displayed correctly in the simulation which I tested by changing stuff in the field.txt file. I checked that it was stunning everything appropriately by using cout on the stunned status of all the actors present on those coordinates of the pools of water. To make sure that they don’t permanently stun an actor, I had to create a Boolean variable which got toggled whenever the actor got stunned by a pool and when it started moving again.

POISON

This was tested in exactly the same way as the food, but instead of increasing health of actors, it reduced health.

ANTHILL

The anthills were being displayed correctly in the simulation, which I tested by moving their positions around in the field.txt file. I tried the simulation with varying number of inputs of bug instructions. I checked whether they were producing ants or not by using a cout line whenever it did, and a cout to tell me their health whenever it changed. This way I also got to know whenever they ate food.

BABY GRASSHOPPER

The baby grasshoppers were moving correctly in the simulation, and they were never in the same spot as a pebble, which I could see in the simulation and double check as well by using an if condition and cout if they ever intersect. Apart from that, I checked that they were eating food by using a cout line whenever their health changed and they ate food. This way, I also got to know when they were supposed to die, and when to expect 100 food on the grid. I kept track of whether they were pool stunned or not. I checked whether they were deleted or not upon death or conversion into adult grasshopper by checking the data structure and looking at the simulation.

ADULT GRASSHOPPER

The testing for this was very similar to the baby grasshopper, but the part which was required to check was the jumping part. This I debugged by marking the radius around the grasshoppers by pool of water items and disabling all other pool of water items. This way, I could see that the area which was checked for pebbles was correct.

ANT

After debugging it for all the cases in similar ways as the adult and baby grasshoppers, I had to basically check that the interpreter was working. This I did by making instructions which included every single possible commands, and testing them individually. I outputted the text stored in the command and changes in the status of the ant and the objects of the world. If anything unnecessary or insufficient was happening, I could see in the cout of the program.