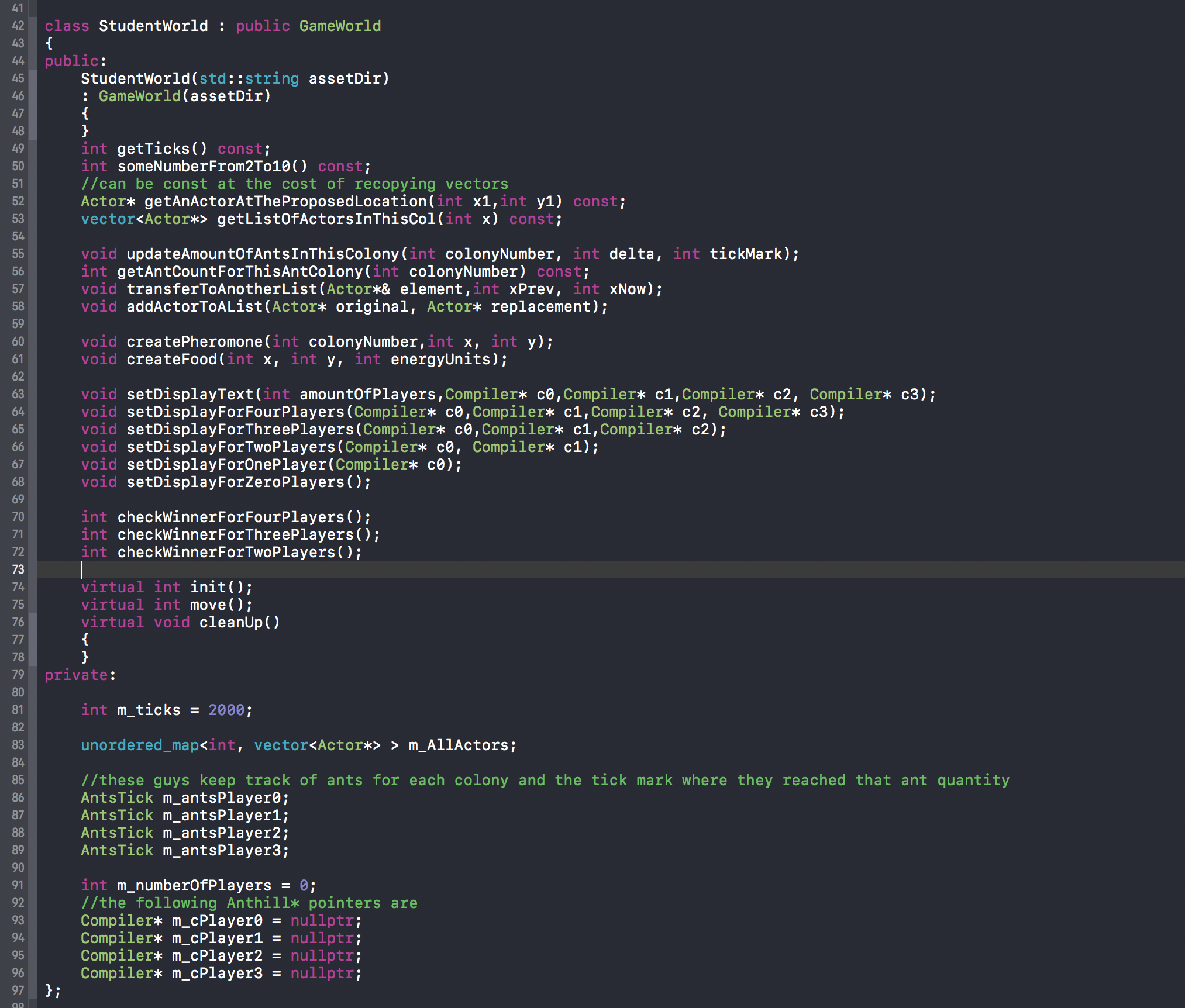
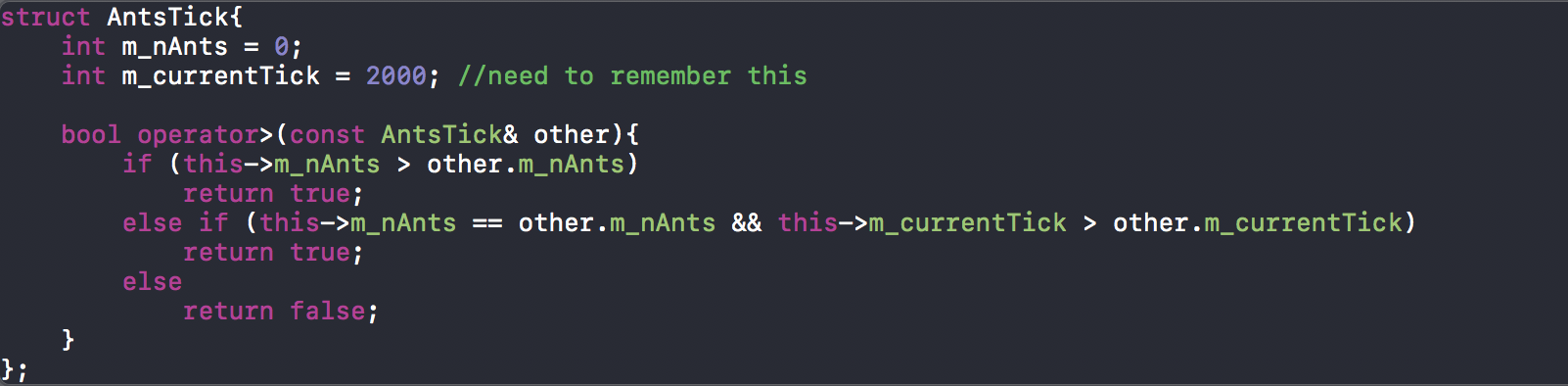
Bugs! Report (Project 3)



Here is the public interface of the StudentWorld class. For any game that contains multiple objects, one of the most important components of the program is storing all the objects and keeping track of their states as they change during the game. I elected to organize all the actors in an unordered map, where the key is a column in the field, and the value being a vector of Actor pointers that are currently in that column. I chose an unordered map because I do not necessarily care about the order of these Actors as they will all be processed regardless of their position in the data structure. Along with recording the ticks that are occuring in the game with **m\_ticks**, StudentWorld records how many players there are in the game with **m\_numberOfPlayers** (anywhere from 0 to 4 players), four Compiler pointers that default to nullptr until a file has been read, and four struct objects called AntsTick.



This struct not only keeps track of all the ants each colony produced, but also the tick at which the ant colony reached that number. I needed this so that in the case of a tie where multiple competitors produced the same additional of ants, I would overload the **> operator** for AntsTick and say that an ant colony is beating another colony if they either have more ants produced or if they tie, the ant colony that reached that number first is greater than the ant colony who produced that number at a later point in the game.

**Some Pseudocode for the Non-trivial StudentWorld Member Functions**

**virtual int init()** {

Seed the pseudo random number generator

Load the field file

Load the scripts for the ant colonies

Create as many Compiler pointers as there are scripts loaded

For every (x,y) position in the field

Get the content at (x,y) from the field file

Store it in main container that holds all actors

Return GWSTATUS\_CONTINUE\_GAME

}

**virtual int move()** {

Decrement ticks

For every alive actor in the game

Tell alive actor to do something

If the actor moved to a different column

Record it so as to transfer it to the vector associated with that column

For every actor that moved to a different column

Transfer it to the appropriate vector associated with that column

For every dead actor in the game

Remove it from the vector it was stored in

Update game text

If ticks in the game is not positive

If someone won

Return GWSTATUS\_PLAYER\_WON

Else

Return GWSTATUS\_NO\_WINNER

Else

Return GWSTATUS\_CONTINUE\_GAME

**virtual void cleanUp()**{

for every column in the game

get the vector of actors in that column

for every actor in that vector

free the memory that those actor pointers are pointing to

}

**Actor\* getAnActorAtTheProposedLocation(int x1, int y1) const** {

Get the vector of actors currently in the column x1;

For every actor in that list

If current actor has same y position as y1

return it;

//otherwise

return nullptr;

}

**vector<Actor\*> getListOfActorsInThisCol(int x) const** {

Use x as the key, and retrieve the vector associated with x;

}

**void transferToAnotherList(Actor\*& element, int xPrev, int xNow)** {

Create a pointer that points to where element is pointing

Add that pointer to the vector associated with xNow

Remove the element from the vector associated with xPrev

}

**void addActorToAList(Actor\* original, Actor\* replacement)** {

add replacement to the same vector that original is in

}

**void setDisplayText(int amountOfPlayers, Compiler\* c0, Compiler\* c1, Compiler\* c2, Compiler\* c3)** {

depending on amountOfPlayers being anywhere from [0,4]

call either setDisplayTextforZeroPlayers(), setDisplayTextforOnePlayer(Compiler\* c0), .. setDisplayTextForTwoPlayers(Compiler\* c0, Compiler\* c1), setDisplayTextForThreePlayers(Compiler\* c0, Compiler\* c1, Compiler\* c2), or setDisplayTextForFourPlayers(Compiler\* c0, Compiler\* c1, Compiler\* c2, Compiler\* c3)

//where all these helper functions display the appropriate amount of ant colonies in game

}

**Other (Trivial) Member Functions of StudentWorld**

**int getTicks() const** returns the amount of ticks in game.

**int someNumberFrom2To10()** returns a number from [2,10]

**void updateAmountOfAntsInThisColony(int colonyNumber, int delta, int tickMark)**  increments the amount of ants in the Anthill with the same colony number as colonyNumber and records when this occurred with tickMark.

**int getAntCountForThisColony(int colonyNumber) const** returns the amount of ants produced by the Anthill with the same colony number as colonyNumber.

**void createPheromone(int colonyNumber, int x, int y)** produces a Pheromone object of the same type as the colonyNumber at location (x,y). If there was already a pheromone there present, it makes the pheromone live longer (up to 768 energy units/health points).

**void createFood(int x, int y, int energyUnits)** creates a Food object at location (x,y) with starting energy units/health points as energyUnits. If there was already a Food object at (x,y), then that Food object would simply gain the energyUnits itself.

**void setDisplayForFourPlayers(Compiler\* c0,Compiler\* c1,Compiler\* c2, Compiler\* c3);**

**void setDisplayForThreePlayers(Compiler\* c0,Compiler\* c1,Compiler\* c2);**

**void setDisplayForTwoPlayers(Compiler\* c0, Compiler\* c1);**

**void setDisplayForOnePlayer(Compiler\* c0);**

**void setDisplayForZeroPlayers();**

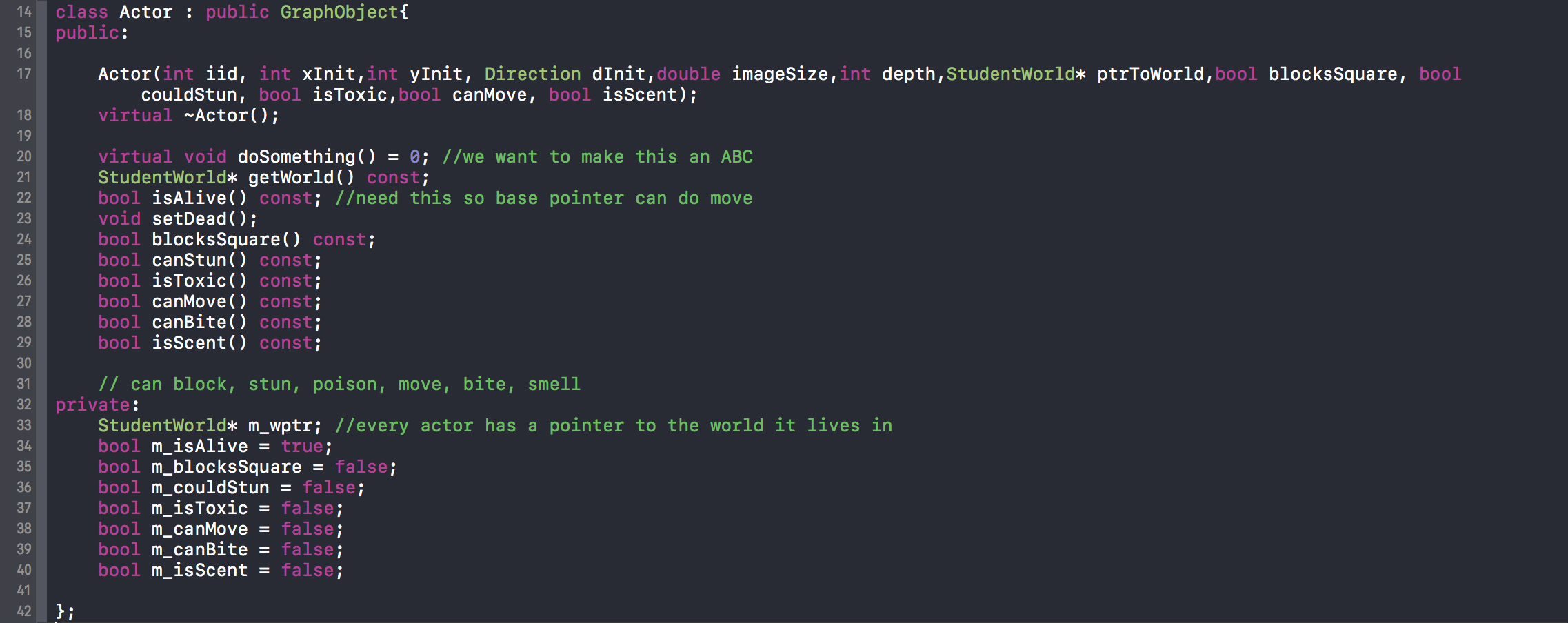
simply display the correctly formatted, appropriate text for the game depending on how ant colonies are in the game.

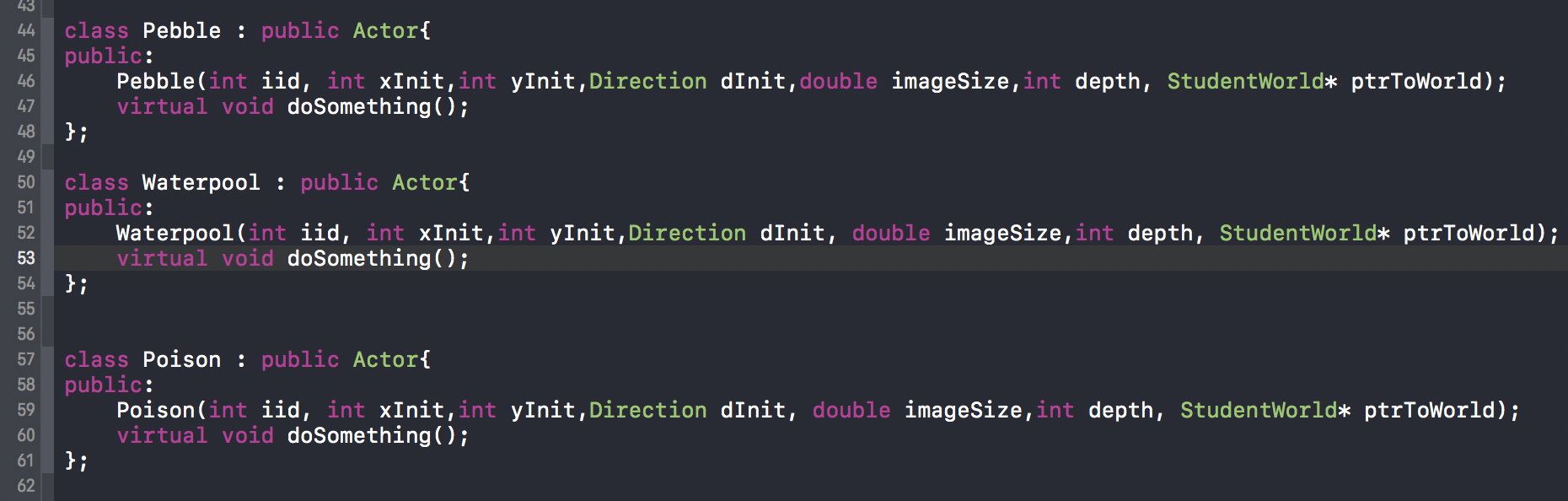
Similarly,**int checkWinnerForFourPlayers()**, **int checkWinnerForThreePlayers()**, and **int checkWinnerForTwoPlayers()** checks for the winner based on how many players there are in the game (for one player that player wins automatically, and there are no winners for zero players).

/////////////////////////////////////////////////////////////////////////////////////////////////////////////

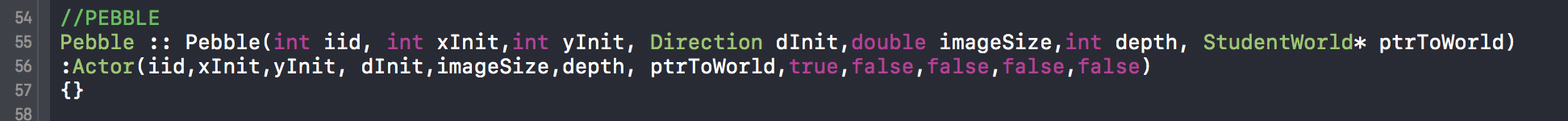
Actor.cpp

////////////////////////////////////////////////////////////////////////////////////////////////////////////



Here are the public member functions and the private member variables for my class Actor. In my project, every actor has certain attributes: **bool isAlive() const** checks if the actor is dead or alive , **void setDead()** sets any actor to a dead state, **bool blocksSquare() const** checks if the actor is capable of blocking other squares or not, **bool canStun() const**  checks if the actor is able to stun or not stun other actors, **bool isToxic() const** checks if the actor could hurt or not hurt other actors because it may be toxic, **bool canMove() const** checks if the actor is able to move during the game or is fixed in position, **bool canBite() const** checks if the actor is capable of hurting other actors or not because it is capable of biting, and **bool isScent() const** checks if the actor has a scent or not so that other actors may smell it. These attributes are passed into the Actor constructor along with its image ID, position in the field, direction that the actor is facing, size of its image in game, depth of its image in game, and a pointer to the world the actor lives in (where **StudentWorld\* getWorld() const** is a function every actor can use to access StudentWorld functions). Although the GraphObject’s destructor is already virtual, it is considered good coding practice to attach the “virtual” keyword in the derived classes’ destructor too. I set **virtual void doSomething() = 0** to be pure virtual because every actor may do something differently, thus creating Actor to be an **abstract base class** in order to prevent the chance of initializing an Actor object. By default, everything starts off alive, and only dead things will be removed from the game. 

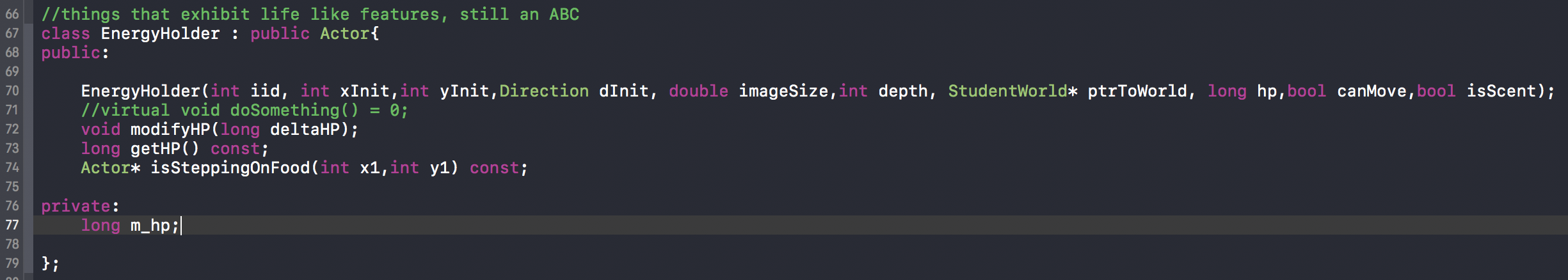
Here are some non-abstract base classes that inherit directly from Actor. All of these actors are fixed in position during the whole game. Pebble has its private member variable **m\_blocksSquare** set to true, Waterpool has its private member variable **m\_couldStun** set to true, and Poison has its private member variable **m\_isToxic** set to true. All of these attributed were passed in their respective initializer lists e.g.



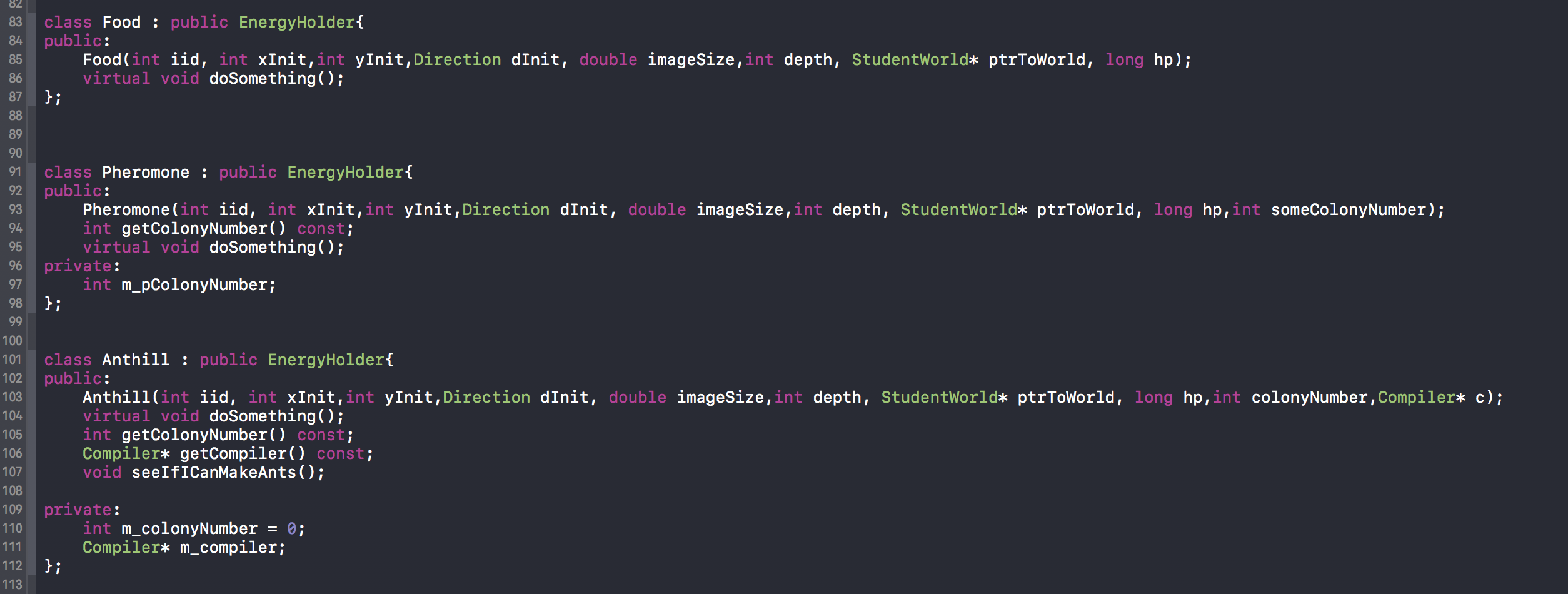
where the first true in that list sets **m\_blocksSquare** to true. Pebble’s **doSomething()** function simply returns. Initially, Waterpool’s and Poison’s **doSomething()** function had implementations, but later in developing my project, I realized that it was much easier (and I am allowed to design this as such) for other actors to step on those actors and trigger an event rather than have Waterpool and Poison wait for something to step on it. There were too many instances where one actor would simply step on the Waterpool, but because it was processed earlier in my container, the actor would move and the Waterpool never had the chance to use its **doSomething()** function to stun it. Consequently,



both of their **doSomething()** functions simply return. So now the other actors have to cause the events that these two classes would have (e.g. if I am an Ant that steps on Waterpool, stun myself)



EnergyHolder is an abstract base class derived from Actor. This class inherits all the public functions of Actor, extends it with the **modifyHP(long delta HP)**, which modifies a new private variable **m\_hp**. The functions **getHP()** **const** simply returns the value of **m\_hp** and **Actor\* isSteppingOnFood(int x1, int y1) const** returns a pointer to a Food object or nullptr otherwise.



These three non-abstract classes derive directly from EnergyHolder. Food’s **doSomething()** simply returns in the same fashion that Waterpool and Poison do, while Pheromone’s **doSomething()** decrements its hp and sets itself to dead if its energy units/health points are non-positive. The Pheromone class also has a **getColonyNumber() const** function that returns the colony number it is associated with.

Anthill is slightly more advanced. It too has a **getColonyNumber() const** but also a **Compiler\* getCompiler() const** function that returns the compiler pointer associated with this ant hill and a **seeIfIcanMakeAnts()** function that determines if the Anthill will produce ants. Its **doSomething()** is capable hurting itself, setting itself to dead, and making new Ants.

//Pseudocode for doSomething

**void Anthill::doSomething()** {

decrement health points by 1

set this to dead if health points are non-positive

If on the same square as food

Take at most 10,000 energy units/health points from that food for a given tick

If current health points >= 2000

Decrement health points by 1500

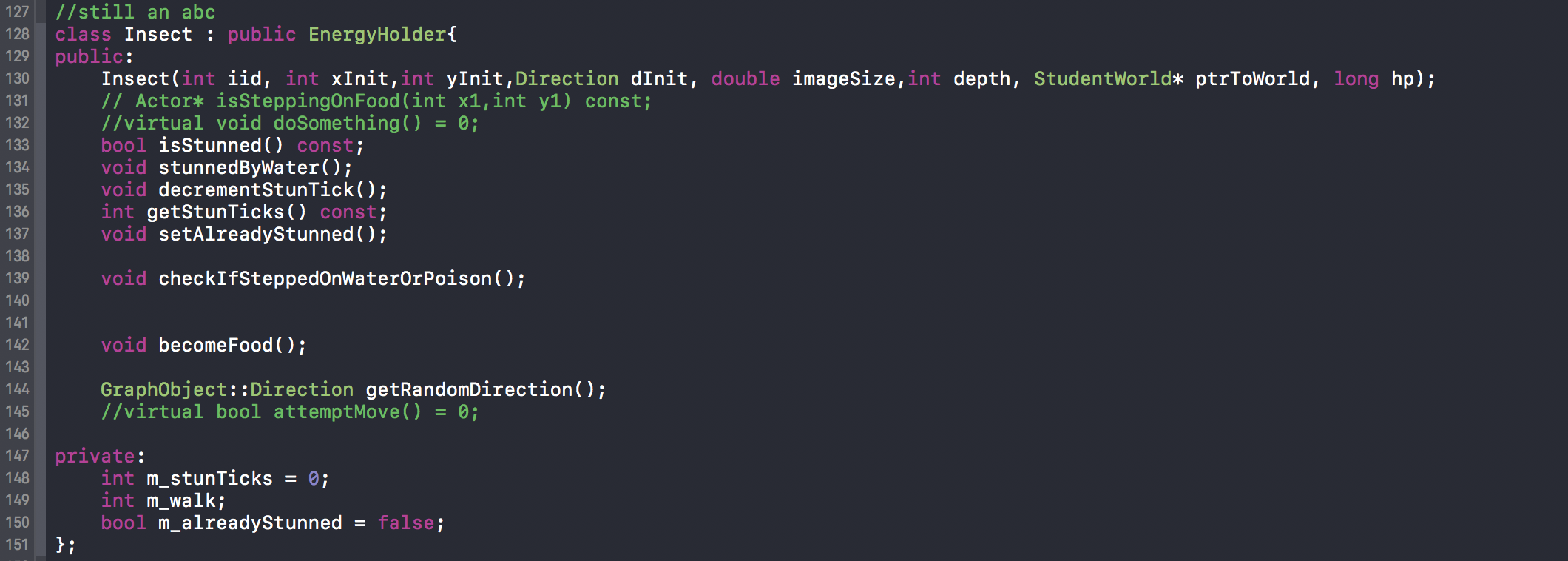
Generate a random direction

Update ant count for this colony

Allocate a new Ant with same colony number as this Anthill and add it to the container

}

//end of Pseudocode



Insect is an abstract base class derived directly from EnergyHolder. They extend EnergyHolder in that all of these objects are capable of moving, being stunned, or become food (unless the spec says otherwise e.g. Adult Grasshoppers derive from Insect and are immune to stuns and poison)

**bool isStunned**() **const** returns true if an Insect is stunned. An Insect is stunned if its **m\_stunTicks** > 0. **void stunnedByWater()** increases **m\_stunTicks** by 2. **void decrementStunTick()** decrements m\_stunTicks by 1. **int getStunTicks() const** returns **m\_stunTicks**. **void setAlreadyStunned()** sets **m\_alreadyStunned** to true, which is important so as to not let any Insect be perpetually stunned. **GraphObject:: Direction getRandomDirection()** generates a random direction. The **checkIfSteppedOnWaterOrPoison()** function simply checks if this insect is currently on the same square as a Waterpool or Poison. **becomeFood()** is a function that either gives an existing Food object more energy units/health points or allocates a new Food object.

//Pseudocode for the slightly more complicated functions

**void Insect :: checkIfSteppedOnWaterOrPoison()** {

Get all actor at the location of this Insect

For every actor in this location

If that actor has **m\_couldStun** set to true

Call stunnedByWater()

Else if that actor has **m\_isToxic** set to true

Decrement health points by 150

}

**void Insect :: becomeFood()** {

Check if currently in same square as a Food object

If is currently on same square as Food object

Increase its energy units/health points by 100

Return

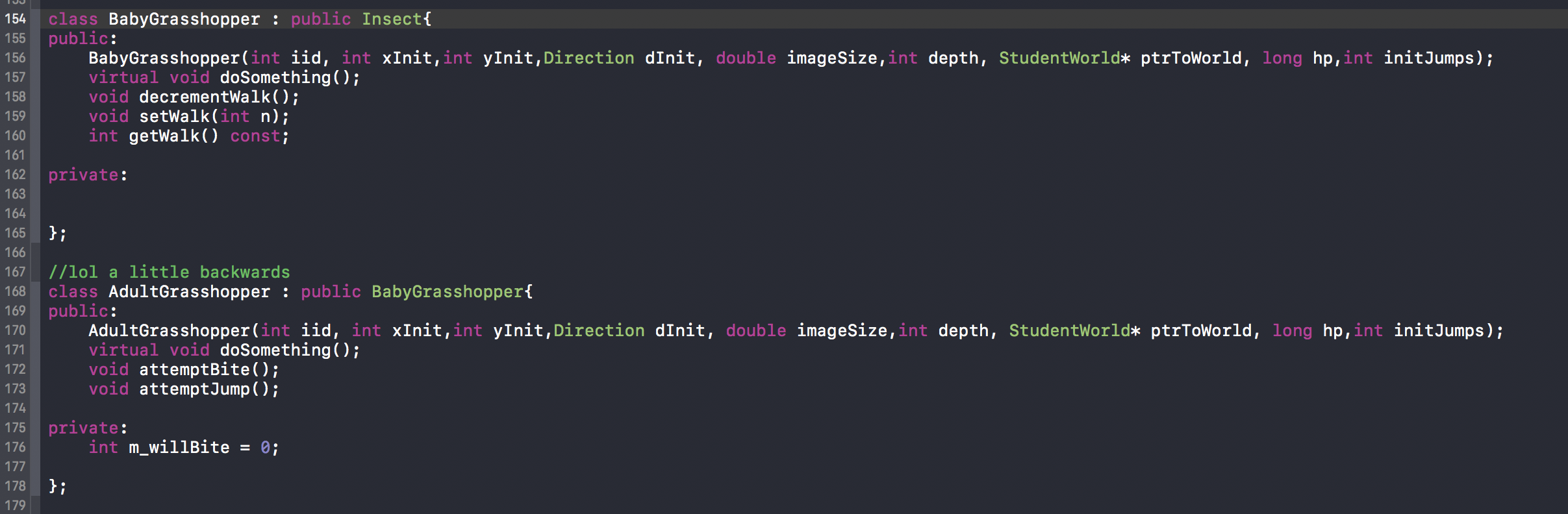
Allocate a new Food object at the same location as this Insect and

Add that Food object to the container of actors

Return

}

//End of pseudocode for the slightly sophisticated functions for Insect



Here is the BabyGrasshopper class that is derived directly from Insect, and AdultGrasshopper derived directly from BabyGrasshopper. Intuitively, the BabyGrasshopper sounds like it should come from the AdultGrasshopper, but when it comes to functionality, the AdultGrasshopper does everything a BabyGrasshopper does, **in addition** to being able to **attemptBite()** and **attemptJump()**. **void decrementWalk()** decrements m\_walk by 1, **setWalk(int n)** sets m\_walk to n, **int getWalk() const** returns **m\_walk**, the amount of steps willing to walk before changing direction. Both of these classes have similar **doSomething()**  functions, which include decrementing its health, setting itself to dead, and eating. The AdultGrasshopper overrides the **doSomething()** function by also jumping and biting.

//Pseudocode for doSomething

**void BabyGrasshopper :: doSomething()** {

decrement HP by 1

Return if on a sleep tick

Set to dead if current HP is non-positive and call **becomeFood()**

If currently stunned

Decrement stunTick

If currently stepping on food

Take at most 200 energy units/ health points from the Food object

If current HP >= 1600

Set this to dead and call **becomeFood()**

Allocate a new AdultGrasshopper at this location

Return

Generate a number from [0,1] //50% chance of resting

If the number is 0

Return

Attempt to move in the direction this BabyGrasshopper is currently facing

If this square is blocked

Set **m\_walk** to zero

If **m\_walk** is positive

Move one square in the direction this BabyGrasshopper is facing

Decrement **m\_walk**

Call **checkIfSteppedOnWaterOrPoison()**

Else

Reset **m\_walk** from any number from [2,10]

Generate a new random direction to face

}//end of pseudo code for slightly complicated functions for BabyGrasshopper

//pseudocode that help describe the slightly more complicated functions for AdultGrasshopper

**void AdultGrasshopper :: attemptBite()** {

If (**m\_willBite** == 1) // 1 in 3 chance of biting

Get a vector of all Insects in the same square as this AdultGrasshopper

If no other Insects is in this square

Return

Generate a random number from [0, vector.size()]

Decrement 50 HP to the Insect at index of random number in vector

If the Insect bitten was an Ant

Have the Ant remember it got bit

If the Insect bitten was an AdultGrasshopper

Generate a random number from [0,1]

If that random number was 1

Have the bitten AdultGrasshopper call **attemptBite()**

}

**void AdultGrasshopper :: attemptJump()** {

Generate a random number from [0,9] // 10% of jumping

If random number generated was 0

Make a float that approximates the mathematical constant Pi (3.14159…)

Get a cosine and sine value from a random radian from [0,2\*pi], and multiply those values by 10

Truncate those values to integers and add those values to current x and y points respectively

Have this AdultGrasshopper move to that final x and y value

**void AdultGrasshopper :: doSomething()** {

decrement HP by 1

Generate a random number from [0,2] and set that value to **m\_willBite**

Return if on a sleep tick

Set to dead if current HP is non-positive and call **becomeFood()**

Call **attemptBite()**

Call **attemptJump()**

If currently stepping on food

Take at most 200 energy units/ health points from the Food object

Generate a number from [0,1] //50% chance of resting

If the number is 0

Return

Attempt to move in the direction this AdultGrasshopper is currently facing

If this square is blocked

Set **m\_walk** to zero

If **m\_walk** is positive

Move one square in the direction this AdultGrasshopper is facing

Decrement **m\_walk**

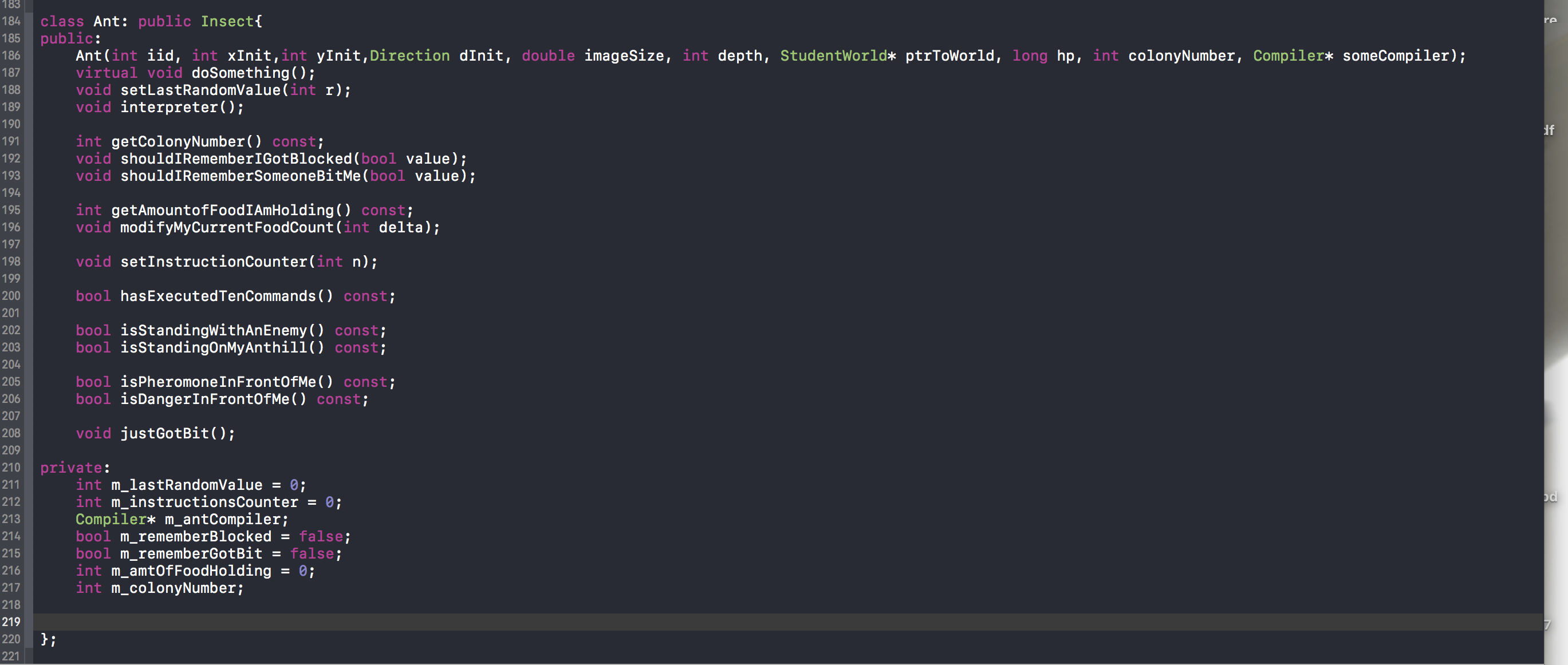
Else

Reset **m\_walk** from any number from [2,10]

Generate a new random direction to face

}

// end of pseudo code for slightly complicated functions in AdultGrasshopper



Here is the Ant class that is derived directly from Insect. **void setLastRandomValue(int r)** sets **m\_lastRandomValue** to r. **int getColonyNumber() const** returns **m\_colonyNumber**, the colony number the ant belongs to. **void shouldIRememberIGotBlocked(bool value)** and **void shouldIRememberSomeoneBitMe(bool value)** are two functions that set **m\_rememberGotBlocked** and **m\_rememberGotBit** respectively to whatever value is. **int getAmountOfFoodIAmHolding() const** returns **m\_amtOfFoodHolding**, the amount of food the Ant is currently holding. **void setInstructionCounter (int n)** sets **m\_instructionsCounter** to whatever n is. **bool hasExecutedTenCommands**() **const** returns true if the Ant has executed ten commands. **bool isStandingWithAnEnemy() const** and **bool isStandingOnMyAnthill() const** are functions that call StudentWorld’s **getListOfActorsInThisCol(int x)** and dynamically casts for Ant of different colony or other Insect for **isStandingWithAnEnemy() const** while **bool isStandingOnMyAnthill() const** dynamically casts for an Anthill of the same colony. Ultimately, these trivial functions are helper functions to the **interpreter()**, which retrieves information from the Compiler pointer and gives the instructions for the Ant to follow.

//Pseudo code that help describe the more complicated functions for Ant

**bool Ant :: isPheromoneInFrontOfMe() const** {

Get the direction this Ant is facing

Depending on this direction, get all the actors in the square the Ant is looking at

Return true if any of those actors is a pheromone of the same colony number as the Ant

//at this point no Pheromone of same type as ant was found

Return false

}

**bool Ant :: isDangerInFrontOfMe() const** {

Get the direction this Ant is facing

Depending on this direction, get all the actors in the square the Ant is looking at

Return true if any of those actors has **m\_isToxic** set to true, is a BabyGrasshopper, AdultGrasshopper, or an Ant of the different colony number as this Ant

//at this point no danger to be found of same type as ant was found

Return false

}

//here comes the heart of the project

**void Ant :: interpreter()** {

make object of type Compiler :: Command

initialize a counter to zero //if this counter goes up to 10, return

Infinite For Loop {

If cannot retrieve a command at line **m\_instructionCounter**

Set this to dead

Return

If command is moveForward

Get direction ant is currently facing

If the square it is looking at is a Pebble

Call **shouldIRememberIGotBlocked(true)**

Else

Call **shouldIRememberIGotBlocked(false)**

Move in the direction the Ant is facing

Call **checkIfSteppedOnWaterOrPoison()**

Go to next command

Increment counter

Return if counter is 10 or more

Else if the command is eatFood

If currently holding more than 100 food

Increase current hp by 100 and decrement food held by 100

Else if holding less than 100 food

Increase current hp by as much food currently held

Decrement food held to zero

Go to next command

Increment counter

Return if counter is 10 or more

Else if the command is dropFood

If not currently holding food

Increment **m\_instructionsCounter**

Increment counter

Return if counter is 10 or more

If dropped food on Anthill

Call Anthill’s **seeIfICanMakeAnts()**

If dropped food on Food

Give that Food object as much energy units as amount of food held

Decrement **m\_amtofFoodHolding** to zero

Else

Allocate a new Food object with as much energy units as food held

Decrement **m\_amtofFoodHolding** to zero

Go to next command

Increment counter

Return if counter is 10 or more

Else if the command was faceRandomDirection

Set this Ant’s direction a randomly generated direction

Go to next command

Increment counter

Return if counter is 10 or more

Else if the command was goto\_command

Set **m\_instructionsCounter** to operand1 of the command object

Else if the command was bite

Get a vector of all insects/Ant of differing ant colony in same square as this Ant

Randomly subtract 15 hp from an element in that vector

Go to next command

Increment counter

Return if counter is 10 or more

Else if the command was pickupFood

If already holding max amount of food

Increment **m\_instructionsCounter**

Increment counter

Return if counter is 10 or more

If stepping on food

Increment **m\_amtOfFoodHolding** by at most 400

Decrement the Food’s energy units/health points

Set the Food to dead if it lost all its energy units

Go to next command

Increment counter

Return if counter is 10 or more

Else if the command was generateRandomNumber

If command’s operand1 is 0

Set **m\_lastRandomValue** to 0

Else

Set **m\_lastRandomValue** to [0, operand1]

Go to next command

Increment counter

Return if counter is 10 or more

Else if the command was getPheromone

Call StudentWorld’s **createPheromone(int colonyNumber, int x, int y)**

Go to next command

Increment counter

Return if counter is 10 or more

Else if the command was rotateClockwise

Get the Ant’s current direction that its facing

Rotate it 90 degrees to the right/clockwise

Go to next command

Increment counter

Return if counter is 10 or more

Else if the command was rotateCounterClockwise

Get the Ant’s current direction that its facing

Rotate it 90 degrees to the left/counter clockwise

Go to next line

Increment counter

Return if counter is 10 or more

Else if the command was if\_command

Get operand1 of the command

If operand1 is “last\_random\_number\_was\_zero”

If **m\_lastRandomValue** is 0

Go to the command at operand2

Else

Go to next command

If operand1 is “i\_am\_carrying\_food”

If **m\_amtOfFoodHolding** is positive

Go to the command at operand2

Else

Go to next command

If operand1 is “i\_am\_hungry”

If Ant’s hp is <=25

Go to the command at operand2

Else

Go to next command

If operand1 is “i\_am\_standing\_with\_an\_enemy”

If calling **isStandingWithAnEnemy()** is true

Go to the command at operand2

Else

Go to next command

If operand1 is “i\_am\_standing\_on\_food”

If calling **isStandingOnFood(int x, int y)** is true

Go to the command at operand2

Else

Go to next command

If operand1 is “i\_am\_standing\_on\_my\_anthill”

If calling **isStandingOnMyAnthill()** is true

Go to the command at operand2

Else

Go to next command

If operand1 is “i\_smell\_pheromone\_in\_front\_of\_me”

If calling **isPheromoneInFrontOfMe()** is true

Go to the command at operand2

Else

Go to next command

If operand1 is “i\_smell\_danger\_in\_front\_of\_me”

If calling **isDangerInFrontOfMe()** is true

Go to the command at operand2

Else

Go to next command

if operand1 is “i\_was\_bit”

If **m\_rememberGotBit** is true

Go to the command at operand2

Else

Go to the next command

if operand1 is “i\_was\_blocked\_from\_moving”

If **m\_rememberBlocked** is true

Go to the command at operand2

Else

Go to the next command

} // END OF interpreter()

**void Ant :: doSomething()** {

decrement its HP by 1

If it’s hp is non-positive

Set it to dead

Call **becomeFood()**

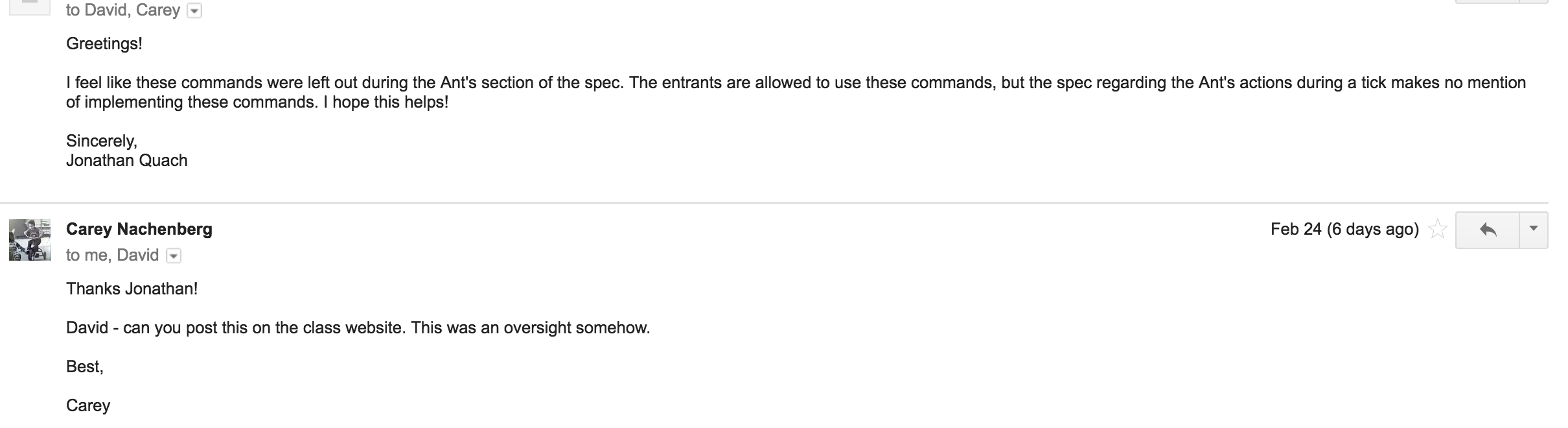
Return

If it’s stunned

Call **decrementStunTick()**

Call **interpreter()**

}//end of pseudo code that summarizes what an Ant does

1. I have no known\* bugs to list. At best, I do know BabyGrasshopper, AdultGrasshopper,and Ant fairly well, but I do not believe these are the same bugs that the spec is asking for. Since the game is definitely playable, along with my very thorough response to **1.**, I think I did an alright job implementing all of the functionalities. Heck, I even found an error in the spec where rotateClockwise and rotateCounterClockwise were not asked to be implemented.

* Of course, having no bugs is different than having no “known” bugs. I was told by Carey that so long as the game is playable and fun (and mine definitely is), then I should be safe.

1. As mentioned earlier, rather than having Waterpool, Poison, and Food call their **doSomething()** if an Ant or BabyGrasshopper stepped on them, the game felt smoother when they simply did nothing and it was up to the Ant or BabyGrasshopper in their **doSomething()** to stun themselves, hurt themselves, or gain their own HP by decreasing the Food’s hp. To reiterate, I chose an unordered\_map as my main container that associates a column with a vector of Actor pointers in that column because order does not matter when iterating through every element anyways as well as being able to insert elements faster. I also have made the assumption that the game should be able to compile and run for as few as zero players and as high as four players.
2. The only tests for Actor, EnergyHolder, and Insect are that my compiler would not allow me to create an instance of any of those objects since all three contained a pure virtual function. Most of testing comes from playtests and using cerr to print out different states of the game.

As for testing StudentWorld, I had it test for bad syntax in Field.txt files, reading in zero Ant programs, one Ant program, two Ant programs, three Ant programs, and four Ant programs. By doing so, I could test that the game would work for 0-4 players. When initializing the Field, I had my compiler print out the size of the unordered\_map, along with the sizes of each vector in the unordered\_map. So seeing that the vector at column[0] has size 64 is comforting because that column is full of Pebbles. Also, I had a BabyGrasshopper start from (1,1) and had it move up and right to (62,62) in an open Field. At the same time, I had my compiler print out that a vector had changed. This happens because every time an Actor leaves a column to another column, one vector decremented in size, and the other vector incremented in size. By doing this, I can feel confident that my container is capable of tracking all of my Actors’ movements and is organizing them properly. Ultimately, I play the game, I see the dead things disappear, and new things appearing. I see the game return GWSTATUS\_NO\_WINNER when no Anthill produced at least 6 Ants, and return GWSTATUS\_PLAYER\_WON when an ant colony produced more than any other ant colony (even in the event of a tie with ants produced, the ant colony that produced those ants the earliest still wins).

Testing Pebble is straightforward. Throughout the game, no Pebble should have moved, and no other Actor is to occupy the same square as it. Simply observing a Field with fewer Pebbles and confirming those two conditions were met should suffice in testing

Testing Waterpool is straightforward. Throughout the game, no Waterpool should have moved, and seeing it stun an Ant or BabyGrasshopper by two ticks, while not stunning AdultGrasshopper are two conditions that needed to be met. Again, creating a Field with only Waterpools and Insects lets me playtest and observe that all the Ants and BabyGrasshoppers were being properly stunned and unstunned while the AdultGrasshoppers were always moving until their death.

Testing Poison is straightforward. Throughout the game, no Poison should have moved, and seeing it hurt an Ant or BabyGrasshopper by 150 HP, while not hurting AdultGrasshopper are two conditions that needed to be met. Like Waterpool, creating a Field with only Poison and Insects lets me playtest and observe that all the Ants and BabyGrasshoppers were being properly hurt while the AdultGrasshoppers would always be the last to survive in the game. I also used cerr to print out the hp of any given insect that stepped on Poison.

Testing Food was alright. I created small Fields by sectioning off areas with Pebbles. I printed out the HP of all the Insects before and after they ate the food to see they were properly gaining HP. Additionally, I printed out the energy units/health points of the food before and after it was being eaten. Moreover, I had many Insects die on Food to make sure it was properly gaining 100 HP from the carcass of the Insect. There is also the case where an Ant would be carrying food, in which, dropping food on a Food would have that Food’s energy units/health point increase exactly as much as the amount of food held by the Ant.

Pheromone had a similar procedure. I created small Fields by sectioning off areas with Pebbles. I printed out the HP of the pheromone every time it was created. I printed out its HP for every tick, observing that it was constantly losing 1 hp for every tick going by. Additionally, I tested the case for dropping a Pheromone on another Pheromone. I created a Pheromone object with 9999 HP and saw it not gain any HP when an Ant released a Pheromone on the same square. I created a Pheromone object with 700 Hp and saw it only gain 68 HP because the maximum HP a Pheromone can have is 768. I also saw the Pheromone object of 400 HP go up to 656 HP when an Ant emitted a Pheromone on that Pheromone object. I also created four Pheromones of differing colonies on the same square because that is legal.

Testing for Anthill is alright. I saw its HP decrement for every tick. I saw it produce over fifty Ants when I gave the Anthill 999999 HP. I see it consistently produce five Ants at the start of every simulation. I saw it produce over fifty Ants when I created a Food of 99999 HP and had an Ant pick it up and drop it on the Anthill, in which, I saw it eat only up to 10000 HP per tick. Similarly, I had an Ant deliver 2600 food to the Anthill, in which it consumed it all in one tick and it produced an Ant almost immediately.

Now, testing for the Insects are a little harder because they move around. For BabyGrasshopper, I had them print out their HP every time they ate to make sure they became AdultGrasshopper when they go past 1600 HP. I had them print out that they were going to die when their HP dropped to zero or lower. I had them tell me when they planned to move or continue sleeping. I saw them get stunned by the Waterpool by 2 ticks and get hurt by 150 HP by poison. I saw them feed a Food by 100 HP when they leave their carcass (either by death or by becoming an AdultGrasshopper). Considering that it is hard to predict their movement, I would say are randomly moving in the [2,10] line that the spec requests. Most of this testing took place in a section of the field filled with Pebbles so I can focus on a few BabyGrasshoppers.

Similarly, For AdultGrasshopper, I had them print out their HP for every tick as I saw it slowly decrease. I forced them to bite other Insects to make sure they are hurting them by the correct amount of HP. I had them tell me when they planned to move or continue sleeping. I had them tell me where they planned to jump in the Field to make sure it was within the Field and was not blocked by a Pebble. I saw them walk through Poison and Waterpool with no effect. I saw them feed a Food by 100 HP when they leave their carcass. Considering that it is even harder to predict their movement since they jump around compared to BabyGrasshopper, I would say they move randomly enough.

Testing for Ants is the hardest. First things first, I made sure they were decrementing their HP by 1 for every tick. I had them tell me they were going to set themselves to death when their HP dropped to zero or lower. I observed them getting stunned by Waterpools and hurt by Poison.

To test their **interpreter()**, I wrote simple scripts. I made a small section for them surrounded by Pebbles. I tested it would die if it cannot retrieve a command (e.g. its instruction counter was negative). I had them attempt to move towards a Pebble and towards open space. I had them eat Food with 10 HP, 100 HP, and 123 HP as Ants are only capable of eating 100 energy units of Food at a time. I had them drop food on food, open space, and on their Anthill. I forced them to bite, and had them print out that they will not bite an Ant of their colony or they printed out they will bite an Insect. I tested if they picked up Food with 10 HP, 400 HP, and 626 HP. I saw them tell me they could not pick up anymore food because they carried 1800 HP of Food already, or they told me they will only pick up enough Food to reach 1800 HP of Food held. I had them emit Pheromones in open space, Pheromones of same colony, and Pheromones of differing colony (just like testing for Food). I had the five initial Ants in the game tell me they will randomly face some direction along with the number they chose to randomly generate using generateRandomNumber. I had Ants spin forever on a forever loop when I wrote goto start in the script and the start had them rotateClockwise (same testing for rotateCounterClockwise) For the if\_command, I simply had them call emitPheromone for every time the condition was met. So for each of the ten possible conditions, I would create one script that met those conditions, in which case the Ant would print to me that the condition was met and it would emit a Pheromone, and one script that failed those conditions, in which case the Ant would print to me that that condition was not meant and it will simply increment its instruction counter.