1.

My Actor inheritance hierarchy:

GraphObject

Actor

Insect

Ant

Grasshopper

AdultGrasshopper

BabyGrasshopper

Anthill

Food

Pheromone

Poison

Water

Pebble

**Actor:**

Functions that allows its subclass to access its private data member

    int getEndurance() const {return m\_endurance;} energy remaining

    int getLoc() const {return m\_locOneD;} transfer 2-D to 1-D

    std::string getStatus() const {return m\_status;}

    int getTurn() const {return m\_turn;} water will increase 2 turns

StudentWorld\* getWorld() const {return m\_studentWorld;}

    bool checkStatus(char st); find if a status is in the string

Functions that allows its subclass to update the private datamember

    void updateTurn(int a = 0) {m\_turn += a;};

    void updateLoc(int x, int y) remain constituency in location

    void updateEndurence(int change) {m\_endurance += change;}

    void updateStatus(char cur) push the latest status into string

    void clearStatus() {m\_status = "";} new location new status

Functions that allow certain functionality of its subclasses to work in the polymorphism design

These are all virtual function and implemented as the general cases in the base class. They also serve as some kinds of identifiers when the actors call studentworld to do things on other classes.

virtual bool blockInsect() {return false;}  only pebble

virtual bool isFood() {return false;}

    virtual bool canBePoisoned() {return false;} ant and babygrasshopper

    virtual bool deadDropFood() {return false;} ant and grasshopper

    virtual bool poisonous() {return false;} poison and water

    virtual int checkColony() {return -1;} anthill, ant, and pheromone has colony number

virtual void biteBack() {}; adultgrasshopper bites back

virtual bool complete() {return false;} ant maybe complete

virtual void notComplete() {}; reset m\_complete for ant

 Pure virtual for doSomething to force every subclass to implement their own version

    virtual void doSomething() = 0;

**Insect:**

No private data member added here, only one public function

int attemptNextStep(Direction dir);

Insects all want to move in their direction, this function uses switch cases for four directions, calls studentWorld to check if next step is possible. If the move is not successful, it will add ‘l’ (blocked) to m\_status to record.

**Ant:**

It has several private members

    int m\_colony (record colony)

    Compiler\* m\_compiler;

    int m\_food; (the food it carries)

    int m\_ic; (instruction counter)

    int m\_lastRandom; (last random number)

    int m\_momX;

    int m\_momY; (its anthill's position)

bool m\_complete;

(because they are pushing around in the data structure, it used to mark this tick has complete)

Trivial functions that implement its traits

    virtual bool canBePoisoned() {return true;}

    virtual bool deadDropFood() {return true;}

    virtual int checkColony() {return m\_colony;}

    virtual bool complete() {return m\_complete;}

virtual void notComplete() {m\_complete = false;}

doSomething function:

    virtual void doSomething();

Generally, it does things specified in spec. m\_ic is set to zero each time to begin at the start line. I put the for loop outside of the interpreter, so that the interpreter only deal with one instruction at a time. I move actors to their new position after one tick of completion except for ant. To avoid recalling its doSomething from another position, after ten time, m\_complete is set to true.

All the other functions, including interpreter(), are private member functions that do their thing.

**Grasshopper:**

It has only one private data member:

int m\_dis;  distance to walk in current direction

Functions for its subclasses baby and adult:

    int getDis() {return m\_dis;}

    void updateDis(int a) {m\_dis += a;}

    virtual bool deadDropFood() {return true;} All its subclass will turn into food after death

    void pickNewWay(); It set new random direction and distance

**BabyGrasshopper:**

Trivial functions of its trait:

virtual bool canBePoisoned() {return true;}

doSomething function:

Generally, it does things specified in spec. It first check endurance, then decrease tick if it’s not zero. It may call studentWorld once to pick food at its location. If it is still awake, it will attempt next move using the Insect class function and then decrease its distance for 1. If it is blocked or finished in current direction it will call grasshopper function to pick a new direction. Finally, it will increase its turn by 2 to make sure it sleep through at least the next two turns.

**AdultGrasshopper:**

Trivial functions of its trait:

virtual void biteBack(); AdultGrasshopper has a 50% chance to bite back, if it does get to bite back, it will call studentWorld to do this of it

doSomething function:

Generally, it does things specified in spec. It is very similar with baby grasshopper except it has a chance to bite something or jump around. A bool finished is used to mark if it is done this round besides increasing turns. Jump around is achieved by first pick a randomX between (-10, 10), calculate the corresponding randomY and pick a random sign. A do-while loop, maximum 2000 times, is used to make sure it does not jump onto a pebble.

**Anthill:**

It has two private data members, which will be passed to the new ant the anthill creates

    int m\_colony;

    Compiler\* m\_compiler;

doSomething function:

Generally, it does things specified in spec. It first ask studentWorld to check and pick whatever food available in its square, then eat the food. When it has enough endurance, it will ask studnetWorld to push a new ant of its colony on its location.

**Food:**

No other private member

Trivial functions of its trait:

    virtual bool isFood() {return true;} for other actors to identify and eat it

doSomething function: It does not do anything

**Pheromone:**

It has one private member for its colony:

    int m\_colony;

Functions:

virtual int checkColony() {return m\_colony;}

virtual void doSomething() {updateEndurence(-1);} it only need to fade away over time

**Poison:**

No private data members

Trivial functions of its trait:

   virtual bool poisonous() {return true;} both poison and its subclass water are poisonous, it is used for ant to check if there is danger ahead

doSomething function:

It asks studentWorld to poison all the insects (that can be poisoned) on its square

**Water:**

No private data members

doSomething function:

It asks studentWorld to stun all the insects on its square

**Pebble:**

No private data members

Trivial functions of its trait:

   virtual bool blockInsect() {return true;} it blocks insect

doSomething function: It does not do anything

**StudentWorld:**

Private members:

    int m\_tick;

    int m\_antNum[4] = {0, 0, 0, 0}; Ant produced in each colony

int m\_winner; Current winner

int m\_players; Player number in the game

    Compiler\* m\_compiler[4]; Compiler for each colony

    std::string m\_bugName[4] = {"", "", "", ""}; Names

std::list<Actor\*> graph[64\*64][2];   It stores a 2-D graph as 1-D array

graph[n][0] is a list of insects (grasshopper and ant)

graph[n][1] is a list of other things (anthill – it can’t be bitten or anything, water, poison, food, and pebble)

Functions for Actors to do things on each other:

1. A function for insects to see if next step is blocked

bool canGo(int x, int y); // check if this step is blocked

It go through all the non-insect objects (graph[x+y\*64][1]) in this square to see if there is one that blockInsects() returns true

1. Functions for actor to perform their features

    bool bite(int x, int y, Actor\* except, int amount);

This function is called by insects that can bite. It takes the location to bite, the exception (insects would not bite themselves), and the amount of damage (ant – 15, adultgrasshopper - 50). Then it go through all the insect objects (graph[x+y\*64][0]) in this square, push the potential enemies who is alive (checkColony() for ants) into a list “enemies”, and then pick a random one to bite. It would add ‘b’ (bitten) to the bitten one’s status. This is for the ant to remember it is bitten on the square.

void stun(int x, int y);    // called by water to stun

This function is called by water. It takes the location to stun. Then it go through all the insect objects (graph[x+y\*64][0]) in this square, add 2 turns on everyone, and add ‘s’ (stunned) into their status string. This make sure it would not stun the insects again before they move away.

void poison(int x, int y);  // called by poison

This function is called by poison. It takes the location to poison. Then it go through all the insect objects (graph[x+y\*64][0]) in this square, subtract 150 endurance from those could be poisoned (ant and babygrasshopper), and add ‘p’ (poisoned) into their status string. This make sure it would not poison the insects again before they move away.

1. Functions to add things into the game

void addAdultGrasshopper(int x, int y);

This function is called by babyGrasshopper when it eat enough food to grow up. It takes the location. Then it push a new AdultGrasshopper into this location (graph[x+y\*64][0]).

void addNewAnt(int x, int y, Compiler\* compiler, int ID, int colony);

This function is called by anthill when it eat enough food to give birth. It takes in the location, pointer to its compiler, specific ID, and its colony. Then it push a new ant into this location (graph[x+y\*64][0]).

    void addPheromone(int x, int y, int colony);

This function is called by ant. It takes in the location and its colony. Then it push a new pheromone of its colony into this location (graph[x+y\*64][1]).

1. Functions about food

bool hasFood(int x, int y, std::list<Actor\*>::iterator &it);

This function is called by ant to check if it is standing on a food. It is also used in the next two funtions. It takes in the location and a reference of an iterator. It goes through all the non-insects in this location (graph[x+y\*64][0]), and set it to the address of the food. It returns whether this location has food or not.

int pickFood(int x, int y, int amount);

This function is called by insects and anthill. It takes in the location and the amount (200 – insects, very large number for anthill to make sure it picks all food). It calls hasFood() and if there is food in this square, it tries to pick the amount, or pick all that exists. It returns the amount of food successfully picked.

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

This function is called by ant and StudentWorld when an insect is dead. It takes in the location and the amount. It calls hasFood() and if there is food in this square, it add the amount to the food’s endurance. It creates a new food object on the square if there is not an existing one.

1. Functions for ants

void updateAnt(int x, int y, int colony);

This function is called by ant after it move itself. It takes in the location and the colony. It go through all the insect objects (graph[x+y\*64][0]) in this square, find the ant of the colony that has changed its recorded location, and push it to the new location’s list. There should be only one ant that is moving each time this function called.

bool hasEnemy(int x, int y, int colony);

This function is called by ant and danger() function. It takes in the location and the colony. It go through all the insect objects (graph[x+y\*64][0]) in this square, and return true once it encounters one insect with different colony number (# - ant, -1 - grasshopper).

bool findPheronmone(int x, int y, int colony);

This function is called by ant. It takes in the location and the colony. It goes through all the non-insect objects (graph[x+y\*64][1]) in this square, and return true once it encounters one object with the same colony number.

bool danger(int x, int y, int colony);

This function is called by ant. It takes in the location and the colony. It first calls hasEnemy() to see if there is other insect enemies in this square, return true if there is. Then it goes through all the non-insect objects (graph[x+y\*64][1]) in this square, to check if they are poisonous (poison and water will return true).

1. Functions to play the game

    int getCurrentTicks() {return m\_tick;}

void updateDisplayText();

This function will update the display text. It check if another colony produces more ants than the current winner, update winner, and create a temporary array of strings. This array copies the names of the ant files, and add ‘\*’ to the winner. Then it create Display in the format the spec requires.

virtual int init()

This function calls the private member function loadField().

    virtual int move()

This function first increases tick by 1. Then it calls all the objects to doSomething(). It updates the list by removing the dead object (and add 100 food for insects), setting to notComplete (for ant to do things in next round), and changing the list for the insects that moved. It then update the displace text and stop the game if the tick reaches 2000.

    virtual void cleanUp()

This function deletes all the dynamically allocated actors in graph and deletes four dynamically allocated compilers.

2. I finished all the functions, and tried my best to eliminate potential bugs.

3. Assumptions:

- The players are less than four, the displace text will be “-----: 00” for the missing players.

- Anthills are not insects but objects. They are pushed into the object list.

- Water is potentially dangerous for ants.

- The ant number is the total produced ants in its colony. The number will not decrease when an ant dies.

- Poison, water and pebble have very large endurance (999999) that they will not possibly die in 2000 ticks.

4. Tests:

Generally, I test the class one by one by commenting out other classes in loadfield() and set break points to trace through steps.

**Pebble:**

I comment out everything besides it and babygrasshopper to see if grasshoppers are blocked.

**Water:**

I comment out everything besides it, pebble and babygrasshopper, go through tick by tick, and count if the babygrasshopper stop four ticks on water.

**Poison:**

I comment out everything besides it, pebble and babygrasshopper, set break point at its doSomething(), and check if it decrease babyGrasshopper’s endurance by 150.

**Food:**

I test it as I tested the insects. I set break points on functions about it to see if they work.

**BabyGrasshopper:**

After make sure the non-insect objects work, I set break point on the place when it should grow up to see if it works. I also change its endurance to make it die very fast to test if food actually is creates at the place it dies.

**AdultGrasshopper:**

After make sure the non-insect objects work, I set break point at the place it bites. I also comment out things to make it jumps every round to test the jump part.

**Anthill:**

To test if it can eat food and bear new ants, I set its initial endurance very large to increase the probability that its ant brings back food. Then I cout when it gives birth to make sure it does bear.

**Ant:**

Ant is built mostly on the functions that already exist. For the new functions and the instruction counter, I test them by setting break points and tracing through. I create four new .bug with slightly different content to see if they do behave differently and to test the display text.

**Pheromone:**

I test this by adding “emitPheromone” into uscant.bug and adding break point.