**1.A high-level description of each class member functions**

Beyond virtual destructors added to each base class with no body, there were also these public functions with each class:

**StudentWorld:**

* **init() -** a virtual int function as detailed by the spec, initializes socrates as well as the correct amounts of bacterial pits, food, and dirtpile, and ensures each does not get created on a space where a prior object is, also as specified by the spec. Though the dirtpiles are allowed to collide with one another as detailed in the spec, for consistency, I made it so no object may be created where it collides with another.
* **move() -** go through each actor alive and give it the chance to move. If Socrates dies, tell the game this and decrease life. If level is finished, play corresponding sound and tell game so. then iterate through actor vector and delete any dead actors, and add Goodies to the playing field as needed. Then modify gameStatText before telling program to continue.
* **cleanUp()** - go through and delete socrates pointer, then any pointers contained in the vector, then erase the whole vector, and ensure the variable of the amount of bacteria left is set to zero.
* **bool AffectBetter(**Actor\* ptr, int distance, bool (Actor::\* f)() const, bool willDamage = false, int amt = 0**) -** allows the user to pass in an actor point, a certain distance, a pointer to a boolean const member function from Actor, and optionally a boolean willDamage to determine if the objects should by damaged and integer amt to determine by how much.

Pseudocode:

Initialize bool test

For every element in the vector of pointers

Temp is the element of the vector we at

If checking for interaction with socrates:

Set variable test to true

Temp points to socrates

Otherwise, if temp points to a dead object or returns false for passed in pointer function:

Proceed to next element in array;

If the calculated distance between object pointed to by ptr and temp is within the passed in distance:

If we are to damage the object pointed to by temp:

Damage it by passed-in amount;

Return true;

If were checking for socrates

Return false;

Return false;

* **double angleToFood(Actor\* ptr) -** pass in pointer of the salmonella object that wants to find angle to closest food in 128 pixels. Returns -1 if no food in range exists. Iterates through vectors of actor pointers and compares minimum distance, and changes the angle to be returned accordingly.

Pseudocode:

Initialize minAngle and minDistance to -1

For every element in the vector of pointers

If the object pointed to by the current element is dead or not food

Proceed to next vector element;

If the distance between the salmonella passed in and food is within 128 pixels

If minDistance is unchanged or the calculated distance is less than it

Change minAngle to the calculated angle between food and the salm

Change minDistance to that calculated distance

Return minAngle;

* **double calcDistance(Actor\* ptr1, Actor\* ptr2, double& diffx, double& diffy )** - passed in two pointers to objects, return the distance between them as well as the differences in their x and y coordinates, respectives as diffx and diffy, for angle calculation. Simple enough for no pseudocode.
* **bool prevOverLap()** - return true if the most recently added object does not overlap with any other objects already in the vector of pointers. False if otherwise.

Pseudocode:

For every element in array except last

If the last added object overlaps with the object the current element is pointing to

Return false;

Return true;

* **void affectBactLeft(int amt) -** modify(+/-) amtofBactLeft variable by passed in amount. NO pseudocode needed, literally one line of code.
* **double calcAngle(double diffX, double diffY) -** calculates angle between two objects passed in the difference in their x and y coordinates using the atan2 function and converts it to degrees. No pseudocode, straightforwards
* **Socrates\* getAPointerToSocrates() -** returns a pointer to the socrates object in studentworld. No pseudo needed. Also Carey said we could do this please don’t dock me points please
* **template<typename Item> void addObj(Item& a) -** generic function to add an object to studentworld’s array of objects, pushes back a new object of the type Item that was passed in, inheriting the x, y, and direction of that Item passed by reference, which is a temporary variable

**Actor:**

* **Actor::Actor -** values passed in and member variable ptrToWorld is set to the passed in pointer to the StudentWorld.
* **virtual bool isAlive() const -** returns status, boolean that represents whether object is alive or dead. Made it virtual as every object has the ability to “die” or get used up to be deleted. Const because does not need to modify anything
* **virtual void setDead()** - sets bool status to false. Every object can “die” so virtual, and only this declaration is needed for it to work on ALL objects.
* **virtual void doSomething()** - is an empty function, does nothing. Thought about making this pure virtual but not only would it give me errors, there were less lines involved when keeping a doSomething that does nothing for objects such as food or dirtpiles, and for every other object will be overridden anyways, as every object has some sort of doSomething that it does or it needs to inherit.
* **virtual StudentWorld\* getAPointerToMyStudentWorld()** - returns a pointer to the StudentWorld object, pretty simple. Virtual as most classes can make use of this so it is useful to pass on.
* **virtual bool BactConsumable() const -** a function to determine whether an object is consumable by bacteria. This version returns false as the only object that is consumable by bacteria is food, which has its specialized version. Virtual to pass this trait on to each object no food. Const because no modifications needed whatsoever to any data.
* **virtual bool Damageable() const -** a function to determine whether an object is damageable by the player’s projectiles. This version returns false, and is virtual and const for much of the same reasons as the previous function.
* **virtual bool canBlock() const -** a function to determine whether an object can block bacteria. Since the only object that can is bacteria it is handy to make all other objects default to false/ virtual and const similar reasons as prior.
* **virtual void Damage(int amt) -** a function to damage any object that does not have HP, like the agents do. This version, which is defaulted by all other objects except agents, runs the setDead function, setting its object to be dead. Virtual so that it is inherited and that the agents can specialize this so they can take an amt of damage.

**DirtPile: public Actor**

* DirtPile constructor - literally just passes in values to actor.
* **Virtual bool Damageable() const -** explained in Actor above, returns true.
* **virtual bool canBlock() const -** explained in Actor above, returns true.

**Pit : public Actor**

* **virtual void doSomething() -** for pit, every tick checks if there is new bacteria to add and if so and the correct randInt is selected, selects a random bacteria to add in alignment with the spec, modifies the amount of bacteria stored by the StudentWorld, and adds the new Bacterium to the StudentWorld through use of the public addObj function

Pseudocode:

If dead

Return;

If random number between 1 and 50 is 1:

Boolean done init to false;

While done is false:

In equal chances between the three bacteria:

If bacteria of selected type is still available, dd new bacteria of that type to StudentWorld and decrease amount of that available bacteria from the Pit’s counts

Then set done to true;

Increase # bacteria in studentWorld through pointer to student world and affectBactLeft;

If no more bacteria let in pit:

Set the pit to dead;

Decreast # bacteria in studentWorld by one;

**ActivatingObject : public Actor**

* **ActivatingObject -** constructor that passes in values to put into integers maxDist and dmgToDeal;
* **virtual void doSomething() -** I was bamboozled by the first provided structure to use for the project and am too far in to turn back, so this function works as a common doSomething for both flame and spray. Please don’t judge me for what seems like crappy organization, anyways here’s what it done do: returns if dead, damages one damageable object if it is overlapping with one then dies, or then moves in its direction and decrements from its distance count. The functions of spray and flame are similar enough that I decided to move their dosomething’s into their common base class as well as their stored variables.

Pseudocode:

If dead:

Return;

If this object overlaps with a damageable object:

Damage it and make this object die;

Then return;

Move this object a certain amount in it’s direction

And decrement maxDist by that amount;

If maxDist is less than or equal to zero:

Set this object to be dead;

**Food : public ActivatingObject**

* **Food::Food -** just a constructor that passes values to the base class
* **virtual bool BactConsumable() const -** explained in Actor group, this version returns true. Because Food can be eaten by bacteria.
* **Virtual void doSomething()** - empty to make sure that food isn’t instantly deleted on launch of the game, as needed because it doesn't do anything.

**Flame : public ActivatingObject**

* **Flame::Flame -** just a constructor that passes value to the base class

**Spray : public ActivatingObject**

* **Spray::Spray -** just a constructor that passes values to the base class

**Goodie : public ActivatingObject**

* **Goodie::Goodie -** a constructor that passes in values and initializes amtToRestore integer based on those values as well as lifetime of goodies based on instructions from spec
* **virtual bool Damageable() const -** all goodies are damageable objects, so this returns true for all of them.
* **virtual void doSomething()** - since the funcitons of all goodies were somewhat similar, I combined them into one in the Goodies class, and differentiate between them based on how much they are to restore the player, whether that be in terms of hit points, life, or flame charges.Gives them a chance to interact with player if they overlap.

Pseudocode:

If dead:

Return;

If this object overlaps with Socrates:

Set this object to be dead;

And Based on the amtToRestore value:

If indicates it is a RestoreHealthGoodie:

Max Socrates’ HP;

Increase score by 250;

If indicates it is a FlameThrowerGoodie:

Max Socrates’ charges through restoreCharges function;

Increase score by 300;

If indicates is an ExtraLifeGoodie:

Increase lives count by one;

Increase score by 500;

If indicates is a Fungus:

Decrease score by 50 points;

Damage Socrates’ HP by 20 points;

If this object is not a fungus:

Play got goodie sound;

Return;

Decrement lifetime by one;

If lifetime is less or equal to 0

Set this object ot be dead;

**RestoreHealthGoodie : public Goodie**

* **RestoreHealthGoodie::RestoreHealthGoodie -** just a constructor that passes in passed in values

**FlameThrowerGoodie : public Goodie**

* **FlameThrowerGoodie::FlameThrowerGoodie -** just a constructor that passes in passed in values

**ExtraLifeGoodie : public Goodie**

* **ExtraLifeGoodie::ExtraLifeGoodie -** just a constructor that passes in passed in values

**Fungus : public Goodie**

* **Fungus::Fungus -** just a constructor that passes in passed in values

**Agent : public Actor**

* **Agent::Agent -** a constructor that passes passed in values and stores values hp for each agent’s hp, dieSound, hurtSound, and a bool isBact. NOTE: I passed in the sound values ONLY to lessen code repeat! I do not differentiate between objects on the basis of those values, only use them in the correct circumstances.
* **int getHealth() const -** simply returns the amount of hp remaining, only used for Socrates but his hp is stored in the agent class so here it is. I do not call it virtual as that is not needed, only this class will make use of it and no derived will specialize. Also const as no data is changed.
* **virtual void Damage(int amt) -** This version of the damage function damages the object by the amount specified, plays a correct sound, check if dead, and if dead plays correct sound. If the object dies and it is a bacteria, the amount of Bacteria left is decreased by one. If health is actually restored, health is set to 100, as done by a restore health goodie.

Pseudocode:

Hp reduced by amt;

If amt was positive:

Play the correct damage sound

If damage killed object:

Play the correct Death sound;

Set this object to be dead;

If the object was a bacteria:

Increase score by 100;

If random digit between 1 and 2 is 1:

Create new Food at spot of Bacteria death and add to StudentWorld;

Decrease amount of bacteria by one;

If amt was negative: (increase in health)

Set hp to 100;

**Socrates : public Agent**

* **Socrates::Socrates -** passes in values to Agent class
* **Virtual void doSomething() -** checks for death, then either moves or shoots watned projectile OR increases m\_numSprays by one each tick, based on user input.

Pseudocode:

If dead:

Return;

If player puts in input and based on user input:

If player wants to move left:

Move the player radially clockwise around the midpoint;

If player wants to move right:

Move the player radially counter-clockwise around the midpoint;

If the player wants to shoot fire and still has charges left:

repeatedly(16 times):

Create a new flame away from Socrates SPRITE\_WIDTH away;

Increase angle to create flame at by 22 degrees;

Decrement the amount of charges by 1;

If the player wants to shoot spray and still has spray left:

Create a new spray object SPRITE\_WIDTH away in front of Socrates;

And decrement m\_numSprays by 1;

Otherwise

If sprays are not at max:

Increase m\_numSprays by 1;

* **int getSprays() const -** non-virtual function becuase unique to Socrates. Returns amount m\_numSprays of sprays left.
* **int getCharges() const -** non-virtual function because unique to Socrates. Returns amount m\_numCharges of charges left
* **void restoreCharge() -** non-virtual function because unique to Socrates. Fills up m\_num charges to max, 5

**Bacterium : public Agent**

* **Bacterium::Bacterium -** passes passed-in values to Agent class and initializes dmg from passed-in int value atk
* **virtual bool Damageable() const -** returns true in this version as all bacteria can be damaged by projectiles, so this is to be inherited. Const as no data is changed.
* **virtual void doSomething() -** contains method that is common to all Bacterium’s doSomething(). Made virtual so that it could be called upon by derived class and specialized. Damages Socrates if the object is overlapping him, or creates a new correct bacteria if 3 food have been eaten by the bacteria, otherwise checks if the bacteria is overlapping a food object and increments its food eaten count after killing it.

Pseudocode:

If overlaps socrates:

Damage him;

Otherwise if 3 food have been eaten:

Create a new bacteria of the same type

Increment StudentWorld’s bacteria counter

Set foodEaten to zero;

Otherwise if overlapping food:

Eat that food

Increment foodEaten by one

**Ecoli : public Bacterium**

* **Ecoli::Ecoli -** passes values to Bacterium class
* **virtual void doSomething() -** if object is alive, runs bacterium’s doSomething , then attempts to move in direction of Socrates. Is virtual so it can utilize code in Bacterium’s doSomething that is common to all Bacteria, which satisfies first few actions Ecoli is to take then speicialized with only Ecoli specific actions.

Pseudocode:

If dead:

Return;

Run bacteria’s doSomething for this object;

If within 256 pixels of Socrates:

Get direction to Socrates;

Then repeatedly up to 10 times:

Try moving towards socrates:

If successful then break loop;

If failed, increment angle by 10 and continue loop

**Salmonella : public Bacterium**

* **Salmonella::Salmonella -** constructor for salmonella class that passes correct values up towards Bacterium class
* **virtual void doSomething()** - virtual so that Salmonella can call Bacterium’s version of doSomething then build off it to make behavior common to both Salmonella sub-types. Also because RegularSalmonella will use this version of doSomething(), which an aggressive salmonella alsu uses and build off of. Calls bacterium’s do something, then attempts to move in current direction as long as its movePlan is not zero. Then otherwise attempts to find the nearest Food object to the Salmonella and change direction towards it. If movement toward that food object fails or no food is found within, then the Salmonella sets its moveplan to 10 and faces a random direction.

Pseudocode:

If dead:

Return;

Call Bacterium’s doSomething();

If movePlan is not zero:

Decrement moveplan by 1;

If movement in current direction is successful:

Complete movement;

Otherwise:

Set moveplan to 10

And set direction to random direction;

Return;

Otherwise

If there is food within 128 pixels:

Set direction towards it;

And if movement towards it is successful:

Complete that movement;

Otherwise:

Set direction to a random degree

And movePlan to 10;

return;

Otherwise:

Set direction to random degreee

And moveplan to 10;

Return;

**RegularSalmonella : public Salmonella**

* **RegularSalmonella::RegularSalmonella -** constructor that passes value to Salmonella class, correct imageID value and such

**AgressiveSalmonella : public Salmonella -**

* **Agressive::AgressiveSalmonella -** passes in correct and needed values to Salmonella class
* **Virtual void AgressiveSalmonella::doSomething()** - virtual to makes use of Bacteriu’s and Salmonella’s doSomething function in certain cases, as well as add its own methods whenever it is called to do something during a tick. If it is not dead, it checks to see if Socrates is within 75 pixels and if so attempts to move in its direction. If this fails, it does not move. After this, it calls Bacterium’s doSomething then returns. Otherwise if it is too far away from Socrates, Salmonella’s doSomething() is called.

Pseudocode:

If dead:

Return;

If socrates within 75 pixels:

And can move 3 pixels in his direction:

Do so;

Call Bacteria’s doSomething();

Return;

Otherwise:

Call Salmonella’s doSomething();

**2. Unfinished Functionalities /Bugs**

I believe I have implemented all functionalities detailed in the spec, however my code seems to have one glaring flaw, the game runs too fast. I am unsure of why this is but its clear that lots more ticks per second are played than in the exe version. This makes my version of the game really hard to test and to play, as it is extremely fast paced!

**3. Design Choices**

Since we were given the option on whether dirt would or could overlap with other dirtpile objects upon initialization in the init function of studentWorld, I simply did not allow this to happen. This leads to may open spaces and allows bacteria to get ore close to Socrates more easily though, and the landscape does look somewhat different from the example exe’s.

I created the functions bactConsumable, Damageable, and canBlock all as boolean virtual in the Actor class to help determine certain actions, such as in my AffectBetterFunciton. This allows dirtpiles to block movement, flames and spray to harm bacteria, and bacteria to consume or locate food.

Since no directions were given on how to calculate an angle between to objects to make one move towards the other, I created a function of my own for this purpose that makes use of the atan2 function.

Since no instructions were given on how to determine if a player has passed a level, I created an integer in StudentWorld’s private members to hold the current amount of Bacteria, and a function that could be called by all Actor objects to modify that amount.

It was also not clear to me from the spec whether the circle of flames protruding from Socrates should begin either at the horizontal or in the direction that Socrates was facing, so I made it similar to the sprays projectiles. In other words the first flame object protrudes in front of Socrates, and the following ones protrude in 22 degree increments from there.

Since it was not really detailed how to initialize objects within a radial distance of the midpoint, I did so by initiliazing them at the midpoint, then move them in a random or specific angle a random or specific distance form the midpoint using the moveAngle method.

**4. How I tested each class (at least the ones that show up on Screen)**

**DirtPile class:**

When I first implemented this, I made sure it existed, and made sure the correct amount were produced by making the game text display the amount of actors (besides Socrates) alive. Upon the implementation of my projectiles, I made sure they were destroyed by the projectiles in one hit. Once I implemented my bacteria I made sure that they could not move if blocked by socrates if blocked by a dirtpile. This was easiest to see in how the salmonella would bounce off the piles, or how Ecoli would simply be unable to move if blocked by a dirtpile in its pursuit of Socrates. Then when implemented the prevention of overlapping on initialization, sure enough no dirt pile overlapped anything else, even other dirtpiles.

**Pit class:**

The first of the functionalities I needed to test was if it correctly created the right amount of each bacterium before dying. To do this, I counted the bacteria as they came out and confirmed that the correct amount were being spawned by the pit. Then I needed to make sure that it was correctly modifying Student World's bacteria counter, so I used the strings writeup so that the screen would display the amount of bacteria at any given tick, and surely, the amount increased with each Bacteria spawned and decreased once the Pit had spawned all it’s contents, and it was dead. Additionally, I made sure that as each Bacteria was spawned, a spawning bacteria sound would play. Once I implemented a method to make sure no initialized object clashed with another I looked at many initializations and confirmed this to be true.

**Food class:**

I made sure first that the food first was not damaged by the player’s projectiles, then confirmed that it did not move by tracking one instance of the Food object by viewing it’s x and y values. Further I made sure that as Bacteria could pass over it freely before I added the functionality for it to be eaten, and once I did I checked that a Bacteria’s counter would increase by one whenever it passed over and ate Food, by displaying this counter. Much like the Pit class I ensured that no Food overlapped with any other object upon initialization.

**Flame class:**

Once I implemented my flame class and could generate flame objects, I made sure that what appeared like a flame around Socrates appeared around Socrates, and that a flame sound would play whenever I pressed the enter button. Then I got close enough to a dirt pile to damage it, and made sure it would disappear once the flame touched it. I did the same with goodies and fungus. Then once I had bacteria successfully basically implemented, I ensured that flames could damage and kill them. I made sure the each separate flame object had the capacity to damage as well. In other words I made sure that more than one object could be harmed at a time within a radius of Socrates if they were close enough to him that each could be harmed by a separate flame object. This goes for dirtpiles as well as bacterium. Also, I made sure each press of the Enter key decreased the amount of charges Socrates still had, and that if no charges were left, another press of enter would not trigger anymore flames or sound.

**Spray Class:**

Much like the flame class, I tested my Spray projectiles first on dirt piles, making sure that just one Spray object would not damage or destroy more than one dirtPile at a time. I went tick by tick to ensure this. Then I tested this on bacterium once they were implemented partly, to make sure they were being damaged properly. Observing how many hits it took to finish certain bacteria due to their health. This way I made sure it was decreasing their hp at the correct rate. Also, I made sure each press of the Space key decreased the amount of sprays Socrates still had, and any subsequent press with 0 sprays would not trigger any creation of a spray objet or sound. However since sprays can only go inward to the circle, I could not test its effectiveness against goodie objects, which the spec says it should be able to damage. However since theoretically the spray could never really reach them, I did not lament this. I also made sure that each spray only lasted a set amount of ticks, based on its maxdist of 112, and that whenever i pressed the button for a spray, it played the correct sound.

**Goodies:**

Since much of Goodie functionality is the same, I decided to test them at nearly the same time. First I made sure that the correct object shows up with the correct image ID’s. Then I made sure that the correct objects had the correct effects upon Socrates, e.g. that the flamerestoregoodie set socrates charges to 5, or the restoreHealthGoodie put him at 100 health. Then I made sure that the positive ones gave him a got goodie sound, while the fungus object did not when made contact with. As mentioned before, I made sure they could be destroyed with flames, and waited them out. Surely enough, each object had a randomly selected lifetime based on the instructions from the spec. Not only this, I observed that the Fungus appeared at a much higher rate than any singular other goodies, which matched up with the instructions from the spec.

**Socrates class:**

To test Socrate’s movement once implemented, I counted how many touches of a button it took to get him what seemed to be an angle of 90 degrees, which at an increment of 5 degrees, was 18 pushes, as expected. I did this for both directions, and ensured he could move around freely around the whole circle. As described in the flame and spray sections, I extensively tested that pressing Enter and space correctly created the flame or spray objects as long as it could. Further, I made sure that if no button was pressed, that the amount of sprays would increase by one if Socrates made no action during a tick.

Like the Goodie section describes, I made sure each Goodie correctly affected Socrates, and only the good ones made the sound got goodie. The harmful fungus, would instead make Socrates play a hurt sound, similarly to Bacteria. I made sure that contact with a Bacteria with Socrates played the correct damaged sound and decremented the health by the correct amount, going tick by tick to observe the decrease per tick, doing so with each type of bacterium.

**RegularSalmonella class:**

In order to test RegularSalmonella’s functionalities specifically I temporarily modified the pit class to only produce Regular Salmonella, removed all Food objects, and observed its behavior. First that it exhibits sporadic behavior, taking on random angles upon each time it collided with a Dirtpile, as well as the fact that it could collide with dirt. Adding in the Food, I saw that regularSalmonella still exhibited sporadic behavior, but moved towards Food slowly but surely, and could eat enough to create another Salmonella! Then I made sure each of my projectiles could damage the Salmonella to make correct damage and hurt sounds, though this was difficult given their sporadic behavior. Through this I was also able to observe they successfully could damage Socrates and could not move past a VIEW\_RADIUS distance from the center of the circle, instead changing its direction and moving that way.

**AgressiveSalmonella class:**

I tested the AgressiveSalmonella’s functionality in a similar method to the RegularSalmonella, ensuring first that it collided with DirtPiles and then that it showed sporadic behavior in the absence of Food and Socrates within a certain distance. Of course once food was introduced, the same behavior observed of RegularSalmonella was seen. I also confirmed its ability to reproduce given enough food and inability to move past the petri dish. The difference was observed that when Socrates was close enough, the AgressiveSalmonella moved only in the direction towards Socrates, or tried to. This made it easier to ensure the AgressiveSalmonella were damageable, and made the correct damage and death sounds.

**Ecoli class:**

Since Ecoli basically dart for the player no matter their or Socrate’s position, it was much more difficult to ensure they could eat food. However , this made it simple to see that dirt objects blocked their movement, and that they were correctly damaged by projectiles, and that when they died they made the correct sounds. Once an Ecoli ate three food, it indeed produced a new Ecoli object as specified. All these effects for each class also let me make sure that a majority of my studentworld functions were functioning as intended.