PUBLIC MEMBER FUNCTIONS

Actor class: (public GraphObject)

**virtual void doSomething() = 0**

This public member function does not do anything. It is in the Actor base class because all actors in the project are able to do something. It is pure virtual because most actors do something unique. Makes the Actor base class abstract.

**bool isAlive()**

This public member function returns whether or not the current actor is alive or not. It is in the Actor base class because all actors may need to know if it’s alive or not. This function is not virtual as all actors can use the same implementation.

**void kill()**

This public member function sets the current actor’s alive status to dead. It is in the Actor base class because all actors may need to be killed. This function is not virtual as all actors can use the same implementation.

**int depth()**

This public member function returns the depth of the current actor. It is in the Actor base class because all actors may need to access depth. This function is not virtual as all actors can use the same implementation.

**void direction(Direction d)**

This public member function returns the direction of the current actor. It is in the Actor base class because all actors may need to access their direction. This function is not virtual as all actors can use the same implementation.

**StudentWorld\* getWorld()**

This public member function returns a pointer to the StudentWorld object in which the actor is populated. It is in the Actor base class because all actors may need to access their StudentWorld object. This function is not virtual as all actors can use the same implementation.

**bool movement(Direction d, int amount)**

This public member function moves an actor in a specified direction by the specified amount (int amount) if possible. It will return true if the movement occurred, or false if an object was obstructing its attempt. It is in the Actor base class because all actors may need to move. This function is not virtual as all actors can use the same implementation.

**virtual bool infectionPossible()**

This public member function returns whether or not the current actor can be infected. It is in the Actor base class because all actors may need to know whether or not they can be infected. This function is virtual because several actors may require that they can be infected (namely Humans). Returns false.

**virtual bool paralysisPossible()**

See infectionPossible(). It has the same functionality except with paralysis.

**virtual bool canOverlap()**

See infectionPossible(). It has the same functionality except with overlapping.

**virtual bool canBurn()**

See infectionPossible(). It has the same functionality except with burning. Returns true.

**virtual bool canExplode()**

See infectionPossible(). It has the same functionality except with exploding.

**virtual bool blocksFlames()**

See infectionPossible(). It has the same functionality except with the ability to block flames.

**virtual bool isDumb()**

See infectionPossible(). It has the same functionality except with checking if the current actor is dumb.

**void infected()**

Infects the current actor. It is in the Actor base class as many other functions require pointers to actors, and I carelessly did not account for this until much too late into the project. This function is not virtual as all relevant actors can use the same implementation. This should be in the Human class.

**void cured()**

See infected(). Same functionality except it cures the current actor.

**bool infectionCheck()**

Returns current infection status. It is in the Actor base class for much of the same reasons as infected() and cured(). This function is not virtual as all relevant actors can use the same implementation.

**void pointValue(int x)**

Adds x amount of points to the player’s total. It is in the Actor base class as many actors may need to adjust points. This function is not virtual as all relevant actors can use the same implementation.

**double distance(double x1, double y1, double x2, double y2)**

Returns the distance between points (x1,y1) and (x2,y2). It is in the Actor base class as calculating distance proves to be quite useful throughout the many actor subclasses. This function is not virtual as all relevant actors can use the same implementation.

**double closestActor(Actor\* a, double x, double y, bool infect, bool para, int type)**

Returns the closest actor of specified type. bool infect and bool para determine the type of actor, namely zombies or citizens. int type determines if the function uses Actor\* a’s coordinates or the parameters double x and double y. It is in the Actor base class as this functionality is useful for several different classes. This function is not virtual as all relevant actors can use the same implementation.

Human class: (public Actor)

**virtual void doSomething() = 0**

See the identically named function in the Actor base class.

**void infectionIncrement()**

Increments the infection count of the current human. This is in the Human class as all humans make use of this functionality. It is not virtual as all relevant actors can use the same implementation.

**int getInfection()**

Returns the infection count of the current human. This is in the Human class as all humans make use of this functionality. It is not virtual as all relevant actors can use the same implementation.

**virtual bool infectionPossible()**

Returns true. It is virtual because it is an overridden version of an identically named function in Actor. This is in the Human class as all humans share this functionality.

**void resetInfection()**

Resets infection count. This should be in the Penelope class, as only Penelope can reset her infection. It is in the Human class mostly due to carelessness.

Penelope class: (public Human)

**virtual void doSomething()**

The Penelope version of the doSomething() function. If Penelope is infected, increment her infection count. If her infection count reaches 500, kill her and return. The player can instruct Penelope through the keyboard; functionalities include moving up, down, left, and right (by 4 pixels each), curing herself by pressing the enter key provided the player has a vaccine, deploying a landmine at her current location provided the player has a landmine, and deploying a flamethrower provided she has gas. The flamethrower will shoot up to three blocks away, provided a wall or exit is not in the way of the flames. This is in the Penelope class as these actions are unique to a Penelope object. It is virtual because it is an overridden version of an identically named function.

**void win()**

Will set completion status to true, allowing for the program to know that the player has beaten the level. This is in the Penelope class as only Penelope can make use of the function. It is not virtual as it does not need to be overridden.

**void landmineAdd(int x)**

Adds int x amount of landmines to Penelope’s arsenal. This is in the Penelope class as only Penelope can do this. This is not virtual as it does not need to be overridden.

**void gasAdd(int x)**

See landmineAdd(int x), but switch “landmines” to “gas.”

**void vaccineAdd(int x)**

See gasAdd(int x), but switch “gas” to “vaccines.”

**void decLandmine()**

Decrements the amount of landmines Penelope has. This is in the Penelope class as only Penelope can do this. This is not virtual as it does not need to be overridden.

**void decGas()**

See decLandmine().

**void decVaccine()**

See decLandmine().

**int daVaccine()**

Returns the amount of vaccines Penelope has. This is in the Penelope class as only Penelope can do this. This is not virtual as it does not need to be overridden.

**int daGas()**

See daVaccine().

**int daMines()**

See daVaccine().

Citizen class (public Human)

**virtual void doSomething()**

The citizen version of doSomething(). If the citizen is dead, return immediately. If the citizen is infected, increment its infection counter. If the infection counter reaches 500, kill the citizen, play the SOUND\_ZOMBIE\_BORN sound, decrease points by 1000, spawn a zombie at the citizen’s coordinates (70% for a dumb zombie, 30% for a smart zombie), and return. If the citizen is paralyzed, switch its paralysis checker and return. Otherwise, set its paralysis checker to true for the next tick, and find the distances between the citizen and the closest zombie and the citizen and the player. If the player is closer or there are no zombies and the player is within 80 pixels of the citizen, the citizen will walk towards the player at two pixels per movable tick. The citizen will run away from any zombies that come within 80 pixels of it, provided the player is not closer. This is in the Citizen class, as only citizens function like this. It is virtual because it is an overridden version of an identically named function in Actor.

**virtual bool paralysisPossible()**

The overridden version of the function. Returns true. In the Citizen class as it is citizen’s version of the function. It is virtual because it is an overridden version of an identically named function in Actor.

**void runAway()**

If the closest zombie is 80 or less pixels away, the citizen will attempt to run in the direction that will get it the furthest away from the zombie. Will stay still if no such direction exists. This is in the Citizen class, as only citizens run away from zombies. It is not virtual as it does not need to be overridden.

Zombie class (public Actor)

**virtual void doSomething() = 0**

See previous pure virtual functions above.

**virtual bool paralysisPossible()**

Returns true. This is the overridden version of paralysisPossible(). It is in the Zombie class as all zombies can be paralyzed. It is virtual because it is an overridden version of an identically named function in Actor.

**void paralysisSwap()**

Swaps the paralysis checker. It is in the Zombie class because all zombies can make use of this functionality. It is not virtual because it does not need to be overridden.

**bool isParalyzed()**

Returns the paralysis checker. It is in the Zombie class because all zombies can make use of this functionality. It is not virtual because it does not need to be overridden.

**int movementPlan()**

Returns the zombie’s movement plan distance. It is in the Zombie class because all zombies can make use of this functionality. It is not virtual because it does not need to be overridden.

**void randomizeMovementPlan()**

Randomizes the zombie’s movement plan by choosing a new movement plan distance between 3 and 10, and choosing a random direction for the zombie. It is in the Zombie class because all zombies can make use of this functionality. It is not virtual because it does not need to be overridden.

**void moveZombie(Direction d)**

Does the same as the movement(Direction d, int amount) function in the Actor base class except moveZombie(Direction d) decrements movement plan distance if it is allowed to move and sets it to zero if it is obstructed. It is in the Zombie class because all zombies can make use of this functionality. It is not virtual because it does not need to be overridden.

**void setMovementPlanDistance(int k)**

Sets movement plan distance to int k. It is in the Zombie class because all zombies can make use of this functionality. It is not virtual because it does not need to be overridden.

**void decMovementPlanDistance()**

Decrements movement plan distance by one. It is in the Zombie class because all zombies can make use of this functionality. It is not virtual because it does not need to be overridden.

**bool doVomit(Actor\* a)**

If a human is within sixteen pixels of the zombie (in the direction the zombie is facing), the zombie has a one in three chance of vomiting at the human. Returns true if the zombie vomits, false otherwise. It is in the Zombie class because all zombies can make use of this functionality. It is not virtual because it does not need to be overridden.

dumbZombie class: (public Zombie)

**virtual void doSomething()**

The dumbZombie version of doSomething(). Returns if dead or paralyzed or successfully vomits. If the movement plan distance is zero, gets a new plan and direction and will attempt to move in the given direction. This is in the dumbZombie class as it is unique to dumbZombie. It is virtual because it is an overridden version of an identically named function in Actor.

**virtual bool isDumb()**

Returns true. In the dumbZombie class as it is unique to dumbZombies. It is virtual because it is an overridden version of an identically named function in Actor.

smartZombie class: (public Zombie)

**virtual void doSomething()**

The smartZombie version of doSomething(). Returns if dead or paralyzed or successfully vomits. If the movement plan distance is zero, gets a new plan. Determine the closest human and the distance to said human. If this distance is greater than 80 pixels, choose a random direction and attempt to move. Otherwise, run towards the human. This is in the smartZombie class as it is unique to smartZombies. It is virtual because it is an overridden version of an identically named function in Actor.

**void runTowards(Actor\* a)**

Attempts to move towards the Actor\* a. This is in the smartZombie class as it is unique to smartZombies. It is not virtual because it does not need to be overridden.

**Actor\* closestHuman(Actor\* a)**

Returns a pointer to the closest human, calculated from Actor\* a. This is in the smartZombie class as it is unique to smartZombies. It is not virtual because it does not need to be overridden.

Wall class: (public Actor)

**virtual void doSomething()**

The Wall version of doSomething(). Do nothing. This is in the Wall class as it is unique to Walls. It is virtual because it is an overridden version of an identically named function in Actor.

**virtual bool canBurn()**

Returns false. In the Wall class as it is unique to Walls. It is virtual because it is an overridden version of an identically named function in Actor.

**virtual bool blocksFlames()**

Returns true. In the Wall class as it is unique to Walls. It is virtual because it is an overridden version of an identically named function in Actor.

Exit class: (public Actor)

**virtual void doSomething()**

The Exit version of doSomething(). Checks if any citizens overlap with itself. If they do, kill the citizen without deducting any points, play the correct sound, increase points by 500, and return. If the player overlaps with the Exit and no citizens are alive, let the player progress and return. This is in the Exit class as it is unique to Exits. It is virtual because it is an overridden version of an identically named function in Actor.

**virtual bool canOverlap()**

Returns true. In the Exit class as it is unique to Exits. It is virtual because it is an overridden version of an identically named function in Actor.

**virtual bool canBurn()**

Returns false. See canOverlap().

**virtual bool blocksFlames()**

See canOverlap().

Thing class: (public Actor)

**virtual void doSomething() = 0**

See previous pure virtual functions above.

**bool isOverlap(Actor\* a, double x, double y, string action)**

Actor\* a points to the Thing calling it. string action determines what the program checks for; “fire” for flames and pits, “infect” for vomit, and “mine” for landmines. Determines if an actor overlaps with coordinates (x, y). If an actor does overlap, and it can be burned for “fire”, can be infected for “infect”, and is not a wall or exit for “mine”, it will do as the spec asks. Will kill the overlapping actor for “fire”, will infect the overlapping actor for “infect”, and will explode for “mine”. Returns true if successfully run, false otherwise. This is in the Thing class as most Things will be able to utilize this well. It is not virtual because it does not need to be overridden.

**virtual bool canOverlap()**

Returns true. In the Thing class as most Things can overlap. It is virtual because it is an overridden version of an identically named function in Actor.

**void mineFunc(Actor\* a, Actor\* theMine)**

Will explode Actor\* theMine, doing as the spec specifies. This is in the Thing class because it is necessary for isOverlap, which is basically the doSomething() for all Things. It is not virtual because it does not need to be overridden.

Pit class: (public Thing)

**virtual void doSomething()**

This is the Pit version of doSomething(). Calls isOverlap from the Thing class. If it returns true, return, if it returns false, check if the player overlaps with the pit. This is in the Pit class as it is unique to Pits. It is virtual because it is an overridden version of an identically named function in Actor.

**virtual bool canBurn()**

Returns false. In the Pit class as it is unique to Pits. It is virtual because it is an overridden version of an identically named function in Actor.

Flame class: (public Thing)

**virtual void doSomething()**

This is the Flame version of doSomething(). Returns if it is dead or it has been alive for two ticks, killing it before doing so for the latter. Increment its tick counter otherwise and see Pit’s doSomething() from here. This is in the Flame class as it is unique to Flames. It is virtual because it is an overridden version of an identically named function in Actor.

**virtual bool canBurn()**

Returns false. In the Flame class as it is unique to Flames. It is virtual because it is an overridden version of an identically named function in Actor.

Vomit class: (public Thing)

**virtual void doSomething()**

This is the Vomit version of doSomething(). See the doSomething() for Flames except call in “infect” when using isOverlap rather than “fire”. This is in the Vomit class as it is unique to Vomit. It is virtual because it is an overridden version of an identically named function in Actor.

**virtual bool canBurn()**

Returns false. In the Vomit class as it is unique to Vomit. It is virtual because it is an overridden version of an identically named function in Actor.

Landmine class: (public Thing)

**virtual void doSomething()**

This is the Landmine version of doSomething(). Returns if it is dead or its safety counter is greater than zero (only after decrementing it though) or if its safety counter reaches zero (only after activating the landmine). Call the isOverlap function with “mine” and return if it returns true. Else, try again for the player. This is in the Landmine class as it is unique to Landmines. It is virtual because it is an overridden version of an identically named function in Actor.

**virtual bool canExplode()**

Returns true. In the Landmine class as it is unique to Landmines. It is virtual because it is an overridden version of an identically named function in Actor.

Goodie class: (public Thing)

**virtual void doSomething()**

The Goodie version of doSomething(). Returns if dead. If the player overlaps with the goodie, reward 50 points, kill the Goodie, play the right sound, and call a pure virtual function pickUp.

This is in the Goodie class because it applies to all Goodies. It is virtual because it is an overridden version of an identically named function in Actor.

**bool isOverlap(double x, double y)**

Checks if the coordinates (x,y) overlap with the player’s current position. This is in the Goodie class because all Goodies can use this implementation, as it is called directly in the doSomething() function. It is not virtual because it does not need to be overridden.

**virtual void pickUp(Penelope\* p) = 0**

See other pure virtual functions above.

**virtual bool canBurn()**

Returns true. In the Goodie class as all Goodies can burn. It is virtual because it is an overridden version of an identically named function in Actor.

vaccineGoodie class: (public Goodie)

**virtual void pickUp(Penelope\* p)**

Adds one vaccine to Penelope’s arsenal. This is in the vaccineGoodie class because it is unique to vaccineGoodies. It is virtual because it is an overridden version of an identically named function in Goodie.

landmineGoodie class: (public Goodie)

**virtual void pickUp(Penelope\* p)**

Adds two landmines to Penelope’s arsenal. This is in the landmineGoodie class because it is unique to landmineGoodies. It is virtual because it is an overridden version of an identically named function in Goodie.

gasGoodie class: (public Goodie)

**virtual void pickUp(Penelope\* p)**

Adds five gas to Penelope’s arsenal. This is in the gasGoodie class because it is unique to gasGoodies. It is virtual because it is an overridden version of an identically named function in Goodie.

StudentWorld class: (public GameWorld)

**virtual int init()**

Builds the world. Dynamically creates all required objects and stores them in a vector.

**virtual int move()**

Calls every Actor’s doSomething(), controls gameplay, ends levels, initializes new levels etc.

**virtual void cleanUp()**

Deletes all dynamically created variables.

**int sizeOf()**

Returns the number of actors excluding Penelope.

**bool actorBlock(Actor\* a, int depth, double x, double y)**

Determines if an actor is blocking the provided (x, y) coordinates.

**bool checkAllOverlap(double x, double y)**

Determines if any actor is blocking the provided (x, y) coordinates.

**bool flameBlock(Actor\* a, int depth, double x, double y)**

Determines if a Flame object can be created at the given (x, y) coordinates.

**~StudentWorld()**

Calls cleanup()

**void addActor(Actor\* a)**

Adds an actor to the vector

**int exitFunc(double x, double y)**

Performs Exit’s functionality

**Penelope\* getPlayer()**

Returns a pointer to the player

**Actor\* getActor(int k)**

Returns a pointer to the Actor at position k

**bool isEveryoneDead()**

Returns true if all citizens are dead

**FAILED FUNCTIONALITIES AND KNOWN BUGS**

I cannot seem to replicate it but ONCE, the SOUND\_CITIZEN\_SAVED sound played for seemingly no reason. This is very strange, as the sound ONLY plays if a citizen has been saved. Perhaps I missed the citizen walking onto the Exit panel, though perhaps I am just tired as it is currently 3:33 AM.

**DESIGN DECISIONS AND ASSUMPTIONS**

It was not specified how flames should be created with mines. The example game provided reveals that the flames created by the mines will overlap with all actors, including walls. I have done so in my code. However, if the restrictions on the flames are to be consistent, simply comment out the current functionality and uncomment the other one and it should work fine.

I implemented several useful functions inside of the StudentWorld class because calling getWorld()->getActor(k) was getting very, very tedious.

I chose to implement the doSomething() functions for pits, flames, vomit, and landmines in the same function, in the interest of saving space.

I assumed that the flame from a flamethrower may cause a vaccine that a zombie may have thrown to not be thrown.

I assumed that placing a mine on a mine would not detonate the either mine.

I assumed that you could place mines on exits, and thus create pits on the exits.

I assumed you could have overlapping pits, even though that makes no sense.

I assumed that I did not have to handle any case in which a mine might be detonated by a goodie.

**TESTING**

Actor:

If the Actor class was broken, the entirety of my code would be garbage. I suppose simply having the code compile and run seemingly correctly should suffice as proof of its functionality.

Penelope:

The Penelope object created by my code can move in all four directions, is blocked by walls, can overlap with objects that can be overlapped with, can die from flames, can be infected, can die from infection, can shoot a line of flame, can have her attempts at shooting a flamethrower stopped by a wall, can use a vaccine if healthy, can use a vaccine to cure infection, can place landmines, decrements all items properly, adds all items properly, can set her win status, and can return the number of items she has properly. This was tested by trying out their functionalities one by one while playing the game.

Citizen:

Testing citizen was a matter of coaxing the idiot to run towards Penelope if Penelope was closer than the nearest zombie. I had citizens run into landmines, spawned zombies near them to infect them, and killed them with my flamethrower. I added a cerr statement at one point to print out a citizen’s infection value and the moment it hit 500 it turned into a zombie where the citizen used to be, and it played the correct sounds for all the above cases. Much of the testing was simply watching the citizen try to escape zombies, I added an extra case into Penelope’s doSomething() for this, where I could manually add smartZombies into the game. After turning a large amount of citizens into zombies, I noticed that there were both dumb and smart zombies, so that part works as well. Killing them decreases the points by the correct amount. These do not stop flames, as seen by shooting through several.

dumbZombie:

I simply walked near the zombie until it turned to me and vomited. I got one stuck facing me and it did not vomit every tick, so the random part is working properly. I set the chances of dropping vaccines to 100% while testing and the zombies do seem to drop them properly. Killing them plays the proper sound and from several minutes of observation, they seem to move around as specified. These do not stop flames, as found by shooting through several. Movement was tested with a cerr statement that printed something out every time it changed its movement plan.

smartZombie:

I had these zombies chase me, and that worked as intended. The vomiting worked as intended as well, killing one returns the right amount of points, they will chase the closest human (tested by running towards it while it was chasing a citizen), and they will move around randomly if no human is near it. Much of this was tested either by killing smartZombies or allowing them to chase me or a human and trying to divert its attention. The vomiting was tested as in dumbZombie. These do not stop flames, as found by shooting through several. Movement was tested with a cerr statement that printed something out every time it changed a target to chase.

Wall:

To be quite honest, there wasn’t much to test. The walls were created where they were supposed to be, and nothing could walk through them, save for the creation of flames on the wall. Flames do not go through walls, except for flames created by mines.

Exit:

I had a bug for a while where zombies could escape through the exit as well, leading to several hours of frustrated scrolling through my code until I ultimately gave up and redid the implementation, after which it worked fine. I tested it by shooting flames in it, towards it, guiding citizens into it, and escaping through it when all citizens were dead. Additionally, I placed mines on it and blew them up, leading to pits on the exits, after which most of the citizens died, though I think that is working as intended. These stop flames.

Pit:

I walked into a pit. I lead a citizen into a pit. I lead a smartZombie into a pit. I watched for minutes until a dumbZombie fell into a pit. I shot fire over a pit. I placed a mine next to a pit and blew it up to see if the flames appear over the pit (they do). I created pits with landmines. The pits work.

Flame:

Everything that is supposed to die to flames dies to flames as tested by shooting them with the flamethrower. I also tested flames created by mines by killing myself, some citizens, and both types of goodies. To test if they disappeared after two ticks, I had a cerr that printed something out every time doSomething() was called. I shot at the exit a couple times, and shot at the walls a couple of times and the flames don’t seem to go through either.

Vomit:

Vomit can infect both citizens and Penelope, as tested by having citizens be infected or by walking into zombies. The vomit ticks were checked in the same manner as the flame ticks.

Landmine:

Landmines were tested by shooting them with flames, setting them near each other and setting them off, and having all movable actors move on them. Note that the mines don’t kill anything, they simply create a flame in its place. Mines were placed on exits and near walls to check the flame functionality and they are working as in the example. They detonate and play the correct sound if and only if a zombie or a human overlap with it.

Goodies:

All goodies were tested by checking the amount of points received, checking the amount of each goodie received, and destroying the goodies with flames. All reasonable actors seem to be able to overlap with them, as intended.