Project 3
Kontagion

For questions about this project, first consult your TA.
If your TA can't help, ask Professor Nachenberg.

Time due:

Part 1: 11 PM Thursday, February 20
Part 2: 11 PM Thursday, February 27

WHEN IN DOUBT ABOUT A REQUIREMENT, YOU WILL NEVER LOSE CREDIT IF YOUR SOLUTION WORKS THE SAME AS OUR POSTED SOLUTION. SO PLEASE DO NOT ASK ABOUT ITEMS WHERE YOU CAN DETERMINE THE PROPER BEHAVIOR ON YOUR OWN FROM OUR SOLUTION!

BACK UP YOUR SOLUTION EVERY 30 MINUTES TO THE CLOUD OR A THUMB DRIVE. WE WILL NOT ACCEPT "MY COMPUTER CRASHED" EXCUSES FOR LATE WORK.

PLEASE THROTTLE THE RATE YOU ASK QUESTIONS TO 1 EMAIL PER DAY! IF YOU'RE SOMEONE WITH LOTS OF QUESTIONS, SAVE THEM UP AND ASK ONCE.

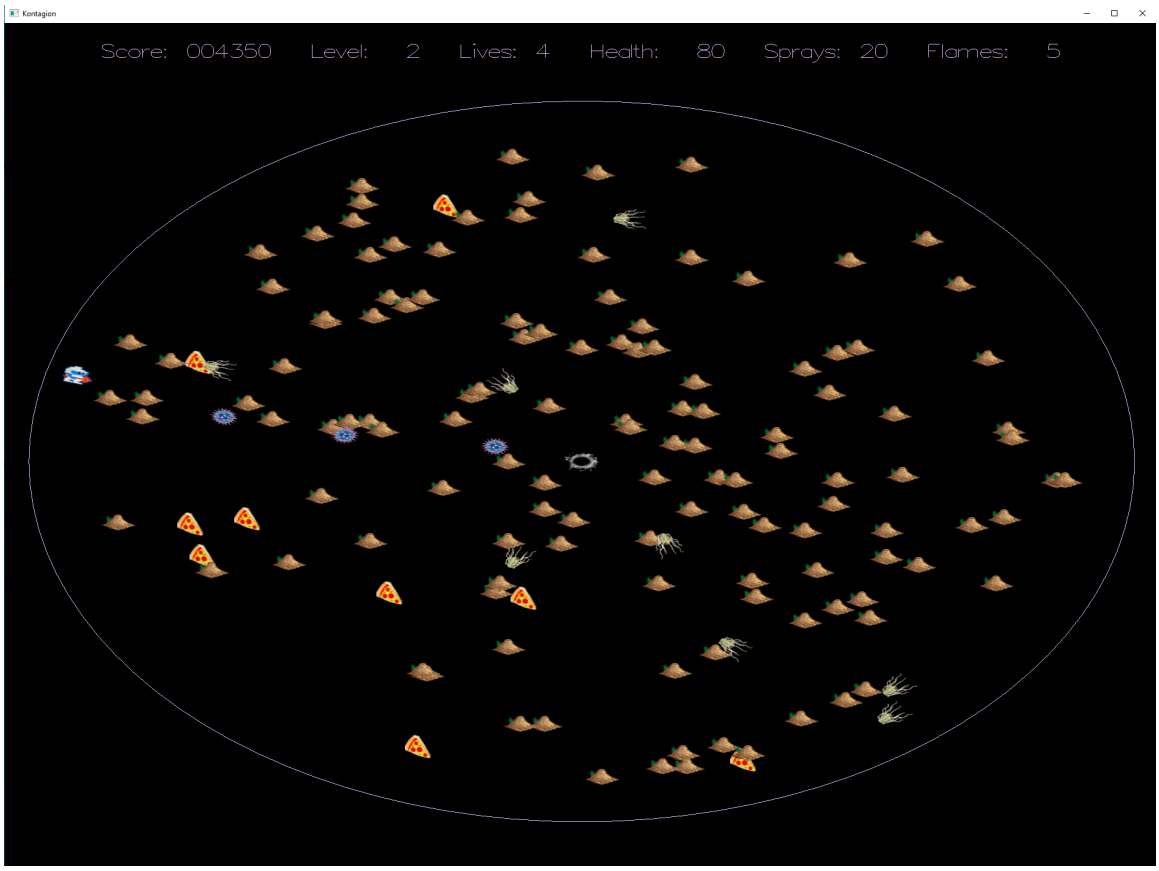
Table of Contents

Introduction

NachenGames corporate spies have learned that SmallSoft is planning to release a new game called Kontagion, and would like you to program an exact copy so NachenGames can beat SmallSoft to the market. To help you, NachenGames corporate spies have managed to steal a prototype Kontagion executable file and several source files from the SmallSoft headquarters, so you can see exactly how your version of the game must work (see posted executable file) and even get a head start on the programming. Of course, such behavior would never be appropriate in real life, but for this project, you'll be a programming villain.

In Kontagion, the notorious Dr. Evyyl (pronounced "EEE vuhl") has been trying to breed trillions of dangerous bacteria so he can release them into the world's public restrooms and wreak havoc. Enter our hero, Socrates Nguyen, an amateur bacteriologist and award-winning bathroom sanitizer. Given his fame killing even the most resistant bacteria, Socrates has agreed to be shrunk to microscopic proportions and deposited within these Petri dishes. There he must exterminate the bacteria before they can be bred and used to decimate the public toilet infrastructure. Each Petri dish has one or more holes in the bottom of it where dangerous bacteria take their naps before coming out to party. Socrates must kill all of the bacteria as they escape from these holes into the Petri dish before the bacteria can feed upon the food in the dish and multiply to epic proportions. Once Socrates has eliminated all of the bacteria from a Petri dish, he advances to the next, more infested, dish where he continues the good fight. There are a near infinite number of Petri dishes, so Socrates will be busy for a long time (or die trying to save the world).

Here's a screenshot of the game:



On the left of the Petri dish, you can see our hero Socrates (pronounced "SOCK ra tease," or "SO crates" for fans of *Bill and Ted's Excellent Adventure*) in his hazmat suit. Socrates can only move clockwise and counterclockwise around the perimeter of the Petri dish – he cannot go into the middle of it. Socrates is equipped with a poison sprayer as well as an omni-directional flamethrower which he can use to battle the deadly bacteria. The Petri dish contains numerous specs of dust (they look like mounds of dirt) which block bacterial movement, as well as microscopic food particles (Dr. Evyyl has found that pizza is a great food source for bacteria). We can also see several different types of bacteria, which multiply as they eat the provided pizza. The blue bacteria are deadly E. coli, and the grey bacteria are various types of salmonella (regular salmonella and nasty, aggressive salmonella). All of the bacteria are deadly, and any time they come into contact with Socrates they will attempt to injure him. In the middle of the Petri dish, we see a pit leading to the area where the bacteria take their afternoon naps (after late night ragers on toilet seats in the frats along Gayley Ave). The bacteria emerge from these hole(s) once they wake up.

While they are not pictured in the image above, the CIA (Central Infection Agency) will occasionally drop "goodies" down for Socrates to use in his battle. The CIA may drop down a health kit (which can restore Socrates to full health), a charge kit to provide five flamethrower charges, or an extra-life kit which will grant Socrates an extra chance to battle the bacteria should he be overwhelmed. If Socrates steps onto these items, he will pick them up. Dr. Evyyl will also occasionally drop deadly fungi into the dish to harm

Socrates; Socrates should stay away from those if he can. From time to time, each of these (good and bad) objects will appear around the perimeter of the dish and will disappear after a short time (they dissolve in the corrosive agar of the dish).

You, the player, will use the following keystrokes to control Socrates:

- Left arrow key or the 'a' key: Moves Socrates counterclockwise
- Right arrow key or the 'd' key: Moves Socrates clockwise
- Space bar: Fires disinfectant spray if Socrates has spray left
- Enter key: Fires the omnidirectional flamethrower if Socrates has flamethrower charges left
- The 'q' key: Quits the game

Points are awarded (or taken away) as follows:

- When Socrates picks up a Restore Health Goodie: 250 points
- When Socrates picks up a Flamethrower Charge Goodie: 300 points
- When Socrates picks up an Extra Life Goodie: 500 points
- When Socrates kills a bacterium of any type: 100 points
- When Socrates comes into contact with a fungus: −50 points

Game Details

In Kontagion, Socrates starts out a new game with three lives and continues to play until all of his lives have been exhausted. There are multiple levels in Kontagion, beginning with level 1 (NOT zero), and during each level the player (aka Socrates) must cleanse the Petri dish of bacteria (including all of the bacteria hidden in the pit(s)) completely before moving on to the next level/dish.

The Kontagion screen is exactly 256 pixels wide by 256 pixels high, although all game activity must occur within a radius of 128 pixels from the center of the screen (within the circle of the petri dish). The bottom-most, leftmost pixel has coordinates x=0,y=0, while the upper-rightmost pixel has coordinate x=255,y=255, where x increases to the right and y increases upward toward the top of the screen. The *GameConstants.h* file we provide defines constants that represent the game's width and height (VIEW_WIDTH and VIEW_HEIGHT), as well as the radius of the Petri dish (VIEW_RADIUS), which you must use in your code instead of hard-coding the integers. Every object in the game (e.g., Socrates, a regular salmonella bacterium, a piece of pizza, etc.) will have an x coordinate in the range [0,VIEW_WIDTH), and a y coordinate in the range [0, VIEW_HEIGHT), and all objects with the exception of flames must be within VIEW_RADIUS pixels of the center of the screen (VIEW_WIDTH/2, VIEW_HEIGHT/2).

Each Petri dish has a random layout composed of one or more bacterial pits, mounds of dirt, and slices of pizza/food. The number of pits, dirt piles and food items varies based

on the level. For instance, easier levels will have more dirt and less food for the bacteria to grow upon. More advanced levels have less dirt to protect Socrates, more food for the bacteria to eat, and more pits containing bacteria. Details will be described in the sections below.

At the beginning of each level, and when the player restarts a level because Socrates died, the level must be reset to an initial random state. That is, all bacteria, mounds of dirt, pits, and slices of pizza must be scattered randomly throughout the dish and freshly initialized (i.e., freshly constructed).

At the beginning of each level, or when the player restarts a level because Socrates died, Socrates starts out with five flamethrower charges, twenty charges of disinfectant, and 100 hit points (health points).

Once a level begins, it is divided into small time periods called *ticks*. There are dozens of ticks per second (to provide smooth animation and gameplay).

During each tick of the game, your program must do the following:

- You must give each object – including Socrates, bacteria, goodies, pits, flames, fungi, etc. - a chance to do something – e.g., fire, move, die, attack, eat, etc.
- You must check to see if Socrates has died. If so, you must indicate this to our game framework (we'll tell you how later) so the level can end and potentially restart fresh, if Socrates has more lives.
- You must delete/remove all dead objects from the game – this includes bacteria that have been destroyed, flames or spray that have dissipated, food that has been eaten, mounds of dirt that have been shot, goodies that have been picked up or dissolved into the agar, fungus that has dissolved or attacked Socrates, etc.
- Your code may also need to introduce one or more new objects into the game – for instance, a new bacterium may be introduced by a pit, a new flame or spray is generated by Socrates when he fires a weapon, a new goodie or fungus can appear, or a dying bacterium may rupture and introduce a slice of pizza into the dish (for other bacteria to gobble up).
- You must update the game statistics line at the top of the screen, including the number of remaining lives Socrates has, the player's current score, the current level number, the number of flamethrower charges and sprays Socrates currently holds, as well as Socrates's current health level (between 0 and 100).
- Check to see if Socrates has completed the current level, and if so, end the current level so Socrates may advance to the next Petri dish.

The status line at the top of the screen must have the following components:

```
Score: 004500  Level: 4  Lives: 3  Health: 82  Sprays: 16  Flames: 4
```

Each labeled value of the status line must be separated from the next by exactly two spaces. For example, the 3 between "Lives: " and "Health:" must have two spaces after it. You may find the *Stringstreams* writeup on the class web site to be helpful.

There are three major types of goodies in Kontagion that Socrates will want to pick up: restore health goodies, flamethrower charge goodies, and extra life goodies. Goodies "trigger" when Socrates step on top of them, update Socrate's stats in some positive way, then disappear. Their respective behaviors are described in the sections on goodies below. Similar to a goodie (but bad for Socrates) are fungi. Like a goodie, a fungus triggers when it comes into contact with Socrates, but instead of doing something good, it drains hit points from him.

There are two major types of projectiles in Kontagion: flames and disinfectant spray. Their respective behaviors are described in the sections on projectile behaviors below.

If Socrates dies (his hit points reach zero) he loses one "life." If, after losing a life, Socrates has one or more remaining lives left, he is placed back in a newly-generated Petri dish at the current level and must again sanitize the entire dish from scratch. If Socrates dies and has no lives left, then the game is over.

Assuming Socrates has spray left, the player may fire Socrates's disinfectant spray towards the center of the Petri dish by pressing the space bar (the spray auto-replenishes over time when Socrates stops firing). If Socrates has flamethrower charges left, he may fire the flamethrower which will shoot flames in all directions around Socrates. Each type of projectile will do different amounts of damage to bacteria, goodies, fungus and dirt that they run into. Food, pits, and fungi are impervious to these projectiles; the projectiles will simply pass over them.

There are three major types of bacteria in Kontagion: regular salmonella, aggressive salmonella and E. coli. Regular salmonella wander around looking for and eating food, and if they happen to contact Socrates they will attack him. Aggressive salmonella will try to move toward Socrates and attack him if he's nearby. Otherwise, they behave just like regular salmonella, hunting for and eating food. E. coli are super aggressive bacteria that can sense Socrates almost anywhere in the Petri dish. They ignore all food (unless they happen to stumble over it) in a never-ending attempt to find and destroy Socrates. All three types of bacteria will undergo cellular division once they've eaten three pieces of food, creating more baddies for Socrates to fight. Just as Socrates can attack, so can bacteria. Their exact behaviors are described in the sections on bacteria behaviors below.

Your game implementation must play various sounds when certain events occur, using the *playSound()* method provided by our *GameWorld* class, e.g.:

```
// Make a sound effect when Socrates fires his flamethrower
pointerToWorld->playSound(SOUND_PLAYER_FIRE);
```

- You must play a SOUND_PLAYER_FIRE sound any time Socrates successfully fires his flamethrower.

- You must play a SOUND_PLAYER_SPRAY sound any time Socrates successfully shoots his disinfectant spray.
- You must play a SOUND_SALMONELLA_HURT sound any time a salmonella of any type is injured.
- You must play a SOUND_SALMONELLA_DIE sound any time a salmonella of any type dies.
- You must play a SOUND_ECOLI_HURT sound any time an E. coli bacterium is injured.
- You must play a SOUND_ECOLI_DIE sound any time an E. coli bacterium dies.
- You must play a SOUND_PLAYER_DIE sound any time Socrates dies.
- You must play a SOUND_BACTERIUM_BORN sound any time a new bacterium emerges from a pit or is created due to another bacterium dividing.
- You must play a SOUND_GOT_GOODIE sound any time the player successfully picks up a goodie.
- You must play a SOUND_PLAYER_HURT sound any time Socrates is injured by a bacterium (when he loses hit points) or a fungus.
- You must play a SOUND_FINISHED_LEVEL sound every time Socrates successfully completes a level.

Constants for each specific sound, e.g., SOUND_BACTERIUM_BORN, may be found in our *GameConstants.h* file.

Determining Object Overlap

In a video game, it's often important to determine if two game objects come into contact with each other (are they close enough that they touch/overlap and therefore interact with one another). For example, if Socrates shoots spray, does this spray come into contact with a nearby bacterium as it flies forward? Or when Socrates walks near a goodie or a fungus, did he get close enough to pick it up?

In Kontagion, the are ***two types of object overlap*** we must check for.

Movement Overlap

When a bacterium tries to move, it will be blocked by a pile of dirt if the Euclidean distance between the center of the dirt and the center of the bacterium's ***target location*** (where it's trying to move to) is less than or equal to 4 pixels (SPRITE_WIDTH/2) apart.

All Other Overlap

Other than movement overlap, two objects are said to overlap if the Euclidean distance between their (x,y) centers is less than or equal to 8 pixels apart (SPRITE_WIDTH). So, for example, in this spec if we say:

- Spray will harm a bacterium, fungus, goodies or a dirt pile if it overlaps with the bacterium/dirt.
- A flame will harm a bacterium, fungus, goodies or dirt pile if it overlaps with the bacterium/dirt.
- A bacterium will eat food if it overlaps with the food.
- Socrates will pick up a goodie if he overlaps with the goodie.
- A bacterium will harm Socrates if it overlaps with Socrates.
- A fungus will harm Socrates if it overlaps with Socrates.

This means that the center (x,y) positions of each of the two objects must be within SPRITE_WIDTH (or 8) pixels of each other for the overlap to occur (using Euclidean distance).

Determining Blocking of Movement

In Kontagion, bacteria can move anywhere on the Petri dish so long as they:

- Don't overlap with a pile of dirt, as described above.
- Stay within the confines of the Petri dish. That is, their (x, y) position must be within VIEW_RADIUS (128) pixels of the center of the dish, which is at:

  (VIEW_WIDTH/2, VIEW_HEIGHT/2)

In Kontagion, Socrates can move anywhere along the rim of the Petri dish – nothing blocks his movement including bacteria, food, fungus, etc. His location can be anywhere along the perimeter of radius VIEW_RADIUS from the center of the screen at:

(VIEW_WIDTH/2, VIEW_HEIGHT/2)

So how does a video game work?

Fundamentally, a video game is composed of a bunch of game objects; in Kontagion, those objects include Socrates, bacteria (e.g., various types of salmonella and E. coli), goodies (e.g., restore health goodies, flamethrower goodies, and extra-life goodies), fungi, projectiles (e.g., flames, spray), dirt, pits, and food. Let's call these objects "actors," since each object is an actor in our video game. Each actor has its own (x, y) location in space, its own internal state (e.g., an aggressive salmonella knows its location, what direction it's moving, etc.) and its own special algorithms that control its actions in the game based on its own state and the state of the other objects in the world. In the case of Socrates, the algorithm that controls the Socrates object is the user's own brain and hand, and the keyboard! In the case of other actors (e.g., a regular salmonella), each object has an internal autonomous algorithm and state that dictates how the object behaves in the game world.

Once a game begins, gameplay is divided into *ticks*. A tick is a unit of time, for example, 50 milliseconds (that's 20 ticks per second).

During a given tick, the game calls upon each object's behavioral algorithm and asks the object to perform its behavior. When asked to perform its behavior, each object's behavioral algorithm must decide what to do and then make a change to the object's state (e.g., move the object 1 pixel to the left), or change other objects' states (e.g., when an aggressive salmonella's algorithm is called by the game, it may determine that Socrates is nearby, and it may try to move toward him). Typically the behavior exhibited by an object during a single tick is limited in order to ensure that the gameplay is smooth and that things don't move too quickly and confuse the player. For example, a salmonella will move just a few pixels forward, rather than moving ten or more pixels per tick; a salmonella moving, say, 20 pixels in a single tick would confuse the user, because humans are used to seeing smooth movement in video games, not jerky shifts.

After the current tick is over and all actors have had a chance to adjust their state (and possibly adjust other actors' states), the graphical framework that we provide animates the actors onto the screen in their new configuration. So if a salmonella changed its location from (10, 50) to (9, 50) (i.e., moved one pixel left), then our game framework would erase the graphic of the salmonella from location (10, 50) on the screen and draw the salmonella's graphic at (9, 50) instead. Since this process (asking actors to do something, then animating them to the screen) happens 20 times per second, the user will see somewhat smooth animation.

Then, the next tick occurs, and each actor's algorithm is again allowed to do something, our framework displays the updated actors on-screen, etc.

Assuming the ticks are quick enough (a fraction of a second), and the actions performed by the objects are subtle enough (i.e., a salmonella doesn't move 3 inches away from where it was during the last tick, but instead moves 1 millimeter away), when you display each of the objects on the screen after each tick, it looks as if each object is performing a continuous series of fluid motions.

A video game can be broken into three different phases:

Initialization: The Game World is initialized and prepared for play. This involves allocating one or more actors (which are C++ objects) and placing them in the game world so that they will appear in the maze.

Gameplay: Gameplay is broken down into a bunch of ticks. During each tick, all of the actors in the game have a chance to do something, and perhaps die. During a tick, new actors may be added to the game and actors who die must be removed from the game world and deleted.

Cleanup: The player has lost a life (but has more lives left), or the player has completed the current level, or the player has lost all of their lives and the game is over. This phase

frees all of the objects in the world (e.g., Socrates, bacteria, pits, flames, sprays, dirt piles, pits, goodies, fungi, food, etc.) since the level has ended. If the game is not over (i.e., the player has more lives), then the game proceeds back to the *Initialization* step, where the level is repopulated with new occupants, and gameplay starts from scratch for the level.

Here is what the main logic of a video game looks like, in pseudocode (The *GameController.cpp* we provide for you has some similar code):

```
while (Socrates has lives left)
{
   Prompt the user to start playing    // "press a key to start"
   Initialize the game world           // you're going to write this

   while (Socrates is still alive)
   {
         // each pass through this loop is a tick (1/20th of a sec)

           // you're going to write code to do the following
         Tell all actors to do something
         Remove any dead actors from the world

           // we write this code to handle the animation for you
         Animate each actor to the screen
         Sleep for one tick to give the user time to react
   }
     // Socrates died – you're going to write this code
   Cleanup all game world objects      // you're going to write this
   if (Socrates has lives left)
      Prompt the player to continue
}

Tell the user the game is over        // we provide this
```

And here is what `Tell all actors to do something` might do:

```
  for each actor on the level:
      if (the actor is still alive)
          tell the actor to doSomething();
```

You will typically use a container (an array, vector, or list) to hold pointers to each of your live actors. Each actor (a C++ object) has a *doSomething( )* member function in which the actor decides what to do. For example, here is some pseudocode showing what a (simplified) regular salmonella might decide to do each time it gets asked to do something:

```
class RegularSalmonella: public SomeOtherClass
{
   public:
       virtual void doSomething()
       {
           If the player overlaps with me in the Petri dish, then
                  Damage the player
           Else if I've eaten enough food to perform cell division
                  Produce a new RegularSalmonella cell adjacent to me
                  Reset my food-eaten count so I don't divide again right away
```

```
            Else if I overlap with food in the Petri dish
                   Eat the food
            Else ...
        }
        ...
};
```

And here's what Socrates's *doSomething( )* member function might look like:

```
class Socrates: public …
{
   public:
        virtual void doSomething()
        {
             Try to get user input (if any is available)
               If the user pressed the LEFT key then
                   Move counterclockwise around the perimeter by 5 degrees
               If the user pressed the RIGHT key then
                   Move clockwise around the perimeter by 5 degrees
               ...
               If the user pressed space and Socrates has Sprays left, then
                  Introduce a new Spray object into the Petri dish in front
                         of Socrates
               ...
        }
        ...
};
```

What Do You Have to Do?

You must create a number of different classes to implement the Kontagion game. Your classes must work properly with our provided classes, and **you must not modify our provided classes or our source files in any way to get your classes to work properly (doing so will result in a score of zero on the entire project!).** Here are the specific classes that you must create:

1. You must create a class called *StudentWorld* that is responsible for keeping track of your game world and all of the actors/objects (Socrates, bacteria, projectiles, goodies, piles of dirt, pits, food, fungi, etc.) that are inside the game.
2. You must create a class to represent Socrates in the game.
3. You must create classes for regular salmonella, aggressive salmonella, E. coli, spray, flames, restore health goodies, flamethrower goodies, extra-life goodies, piles of dirt, pits, fungi, food, etc., as well as any additional base classes (e.g., a bacterium base class if you find it convenient) that help you implement your actors.

You Have to Create the StudentWorld Class

Your *StudentWorld* class is responsible for orchestrating virtually all gameplay – it keeps track of the entire game world (each level and all of its inhabitants such as salmonella, E. coli, Socrates, goodies, projectiles, pits, etc.). It is responsible for initializing the game world at the start of the game, asking all the actors to do something during each tick of

the game, destroying an actor when it disappears (e.g., a bacterium dies, Socrates shoots a flame at a pile of dirt and destroys it, a flame dissipates after flying 32 pixels through the air, etc.), and destroying *all* of the actors in the game world when the user loses a life or advances to the next level.

Your StudentWorld class **must** be derived from our *GameWorld* class (found in *GameWorld.h*) and **must** implement at least these three methods (which are defined as pure virtual in our *GameWorld* class):

```
virtual int init() = 0;
virtual int move() = 0;
virtual void cleanUp() = 0;
```

The code that you write must *never* call any of these three functions (except that *StudentWorld*'s destructor may call *cleanUp()*). Instead, our provided game framework will call these functions for you. So you have to implement them correctly, but you won't ever call them yourself in your code (except in the one place noted above).

Each time a level starts, our game framework will call the *init()* method that you defined in your *StudentWorld* class. You don't call this function; instead, our provided framework code calls it for you.

The *init()* method is responsible for constructing a representation of the current level in your StudentWorld object and populating it with initial objects (e.g., pits, food, dirt piles and Socrates), using one or more data structures that you come up with.

The *init()* method is automatically called by our provided code either (a) when the game first starts, (b) when the player completes the current level and advances to a new level (that needs to be initialized), or (c) when the user loses a life (but has more lives left) and the game is ready to restart at the current level.

After the *init()* method finishes initializing your data structures/objects for the current level, it **must** return `GWSTATUS_CONTINUE_GAME`.

Once a level has been prepared with a call to the *init()* method, our game framework will repeatedly call the *StudentWorld's move()* method, at a rate of roughly 20 times per second. Each time the *move()* method is called, it must run a single tick of the game. This means that it is responsible for asking each of the game actors (e.g., Socrates, each bacterium, each goodie, each projectile, food, fungus, pit, etc.) to try to do something: e.g., move themselves and/or perform their specified behavior. This method might also introduce new actors into the game, for instance adding a new flamethrower goodie into the Petri dish. Finally, this method is responsible for disposing of (i.e., deleting) actors that need to disappear during a given tick (e.g., spray that falls to the ground and disappears, a dead salmonella, etc.). For example, if a salmonella is shot by the Socrates's flames, then its state should be set to dead, and then after all of the live actors in the game get a chance to do something during the tick, the *move()* method should remove that salmonella from the game world (by deleting its object and removing any reference to the

object from the *StudentWorld's* data structures). The *move()* method will automatically be called once during each tick of the game by our provided game framework. You will never call the *move()* method yourself.

The *cleanup()* method is called by our framework when Socrates completes the current level or loses a life. The *cleanup()* method is responsible for freeing all actors (e.g., all bacteria objects, all pit objects, all projectile objects, all goodie objects, the Socrates object, food objects, fungi objects, etc.) that are currently in the game. This includes all actors created during either the *init()* method or introduced during subsequent gameplay by the actors in the game (e.g., a flame that was added to the screen by an exploding food, a goodie that was added after a few minutes of play) that have not yet been removed from the game.

You may add as many other public/private member functions or private data members to your *StudentWorld* class as you like (in addition to the above three member functions, which you *must* implement). You must **not** add any public data members.

Your *StudentWorld* class must be derived from our *GameWorld* class. Our *GameWorld* class provides the following methods for your use:

```
unsigned int getLevel() const;
unsigned int getLives() const;
void decLives();
void incLives();
unsigned int getScore() const;
void increaseScore(unsigned int howMuch);
void setGameStatText(string text);
bool getKey(int& value);
void playSound(int soundID);
```

getLevel() can be used to determine the current level number.

getLives() can be used to determine how many lives Socrates has left.

decLives() reduces the number of lives Socrates has by one.

incLives() increases the number of lives Socrates has by one.

getScore() can be used to determine Socrates's current score.

increaseScore() is used by a *StudentWorld* object (or your other classes) to increase or decrease the user's score upon successfully destroying a bacterium, picking up a goodie of some sort or getting injured by a fungus. When your code calls this method, you must specify how many points the user gets (e.g., 100 points for destroying a salmonella, −50 points if Socrates steps onto a fungus). This means that the game score is controlled by our *GameWorld* object – you *must not* maintain your own score data member in your own classes.

The *setGameStatText()* method is used to specify what text is displayed at the top of the game screen, e.g.:

```
Score: 004500  Level: 4  Lives: 3  health: 82  Sprays: 16  Flames: 4
```

getKey() can be used to determine if the user has hit a key on the keyboard to move Socrates or to fire a projectile. This method returns true if the user hit a key during the current tick, and false otherwise (if the user did not hit any key during this tick). The only argument to this method is a variable that will be set to the key that was pressed by the user (if any key was pressed). If the function returns true, the argument will be set to one of the following values (defined in *GameConstants.h*):

```
KEY_PRESS_LEFT
KEY_PRESS_RIGHT
KEY_PRESS_UP
KEY_PRESS_DOWN
KEY_PRESS_SPACE
KEY_PRESS_TAB
KEY_PRESS_ENTER
```

The *playSound()* method can be used to play a sound effect when an important event happens during the game (e.g., a bacterium dies or Socrates picks up a goodie). You can find constants (e.g., SOUND_PLAYER_SPRAY) that describe what noise to make in the *GameConstants.h* file. The *playSound()* method is defined in our GameWorld class, which you will use as the base class for your StudentWorld class. Here's how this method might be used:

```
// if a salmonella dies, make a dying sound

if (theSalmonellaHasDied())
    pointerToWorld->playSound(SOUND_SALMONELLA_DIE);
```

init() Details

Your StudentWorld's *init()* member function must:

1. Initialize the data structures used to keep track of your game's world.
2. Allocate and insert a Socrates object into the game world. Every time a level starts or restarts, Socrates starts out fully initialized (with the baseline number of sprays, flamethrower charges, hit points, etc.).
3. Allocate and insert various piles of dirt, pits, and food objects into the game world as described below.

Your *init()* method must construct a representation of your world and store this in a *StudentWorld* object. It is **required** that you keep track of all of the actors (e.g., bacteria like aggressive salmonella, pits, foods, projectiles like spray, goodies, etc.) in a **single** STL collection such as a *list*, *set*, or *vector*. (To do so, we recommend using a container of pointers to the actors). If you like, your *StudentWorld* object may keep a separate

pointer to a Socrates object rather than keeping a pointer to that object in the container with the other actor pointers; Socrates is the **only** actor pointer allowed to not be stored in the single actor container. The *init()* method may also initialize any other *StudentWorld* member variables it needs, such as the number of remaining bacteria that need to be destroyed on this level before Socrates can advance to the next level.

You must not call the *init()* method yourself. Instead, this method will be called by our framework code when it's time for a new game to start (or when the player completes a level or needs to restart a level).

Here is how the *init()* method **must** add objects to the Petri dish if the player has reached level number L (L=1 for the starting level):

1. Create and add a new Socrates/player object at location (0, VIEW_HEIGHT/2) to the Petri dish; this is in the left-middle of the dish.
2. Add L pits to the Petri dish at random locations, in a manner such that **no two pits overlap with each other** (their centers must be more than SPRITE_WIDTH pixels apart from each other). Each pit must be no more than 120 pixels from the center of the Petri dish which is at (VIEW_WIDTH/2, VIEW_HEIGHT/2).
3. Add *min(5 * L, 25)* food objects to the Petri dish at random locations, in a manner such that **no two food objects overlap with each other or previously-placed pits** (their centers must be more than SPRITE_WIDTH pixels apart from each other). Each food object must be no more than 120 pixels from the center of the Petri dish which is at (VIEW_WIDTH/2, VIEW_HEIGHT/2).
4. Add *max(180 – 20 * L, 20)* dirt objects to the Petri dish at random locations, in a manner such that **no dirt objects overlap with previously-placed food objects or pits** (their centers must be more than SPRITE_WIDTH pixels apart from each other). **It is OK for dirt objects to overlap with each other, however.** Each dirt object must be no more than 120 pixels from the center of the Petri dish which is at (VIEW_WIDTH/2, VIEW_HEIGHT/2).

move() Details

The *move()* method must perform the following activities:

1. It must ask all of the actors that are currently active in the game world to do something (e.g., ask a salmonella to move itself, ask a pit to introduce a new bacterium into the Petri dish, ask a goodie to check if it overlaps with Socrates, and if so, grant him a special power, give Socrates a chance to move or fire, etc.).
    a. If an actor does something that causes Socrates to die (e.g., an E. coli damages Socrates so that his hit points/health reaches zero), then the *move()* method should immediately return `GWSTATUS_PLAYER_DIED`.
    b. Otherwise, if Socrates has cleared the current Petri dish of all bacteria *and* all of the pits have disappeared (each pit disappears once it's released all of its bacteria into the Petri dish) then it's time to advance to the next

level. In this case, the *move()* method must return a value of `GWSTATUS_FINISHED_LEVEL`.

2. It must then delete any actors that have died during this tick (e.g., a salmonella that was killed by flames and so should be removed from the game world, or a goodie that disappeared because it overlapped with Socrates and activated).
3. It must then add any new objects to the game (e.g., a new goodie or fungus).
4. It must update the status text on the top of the screen with the latest information (e.g., the user's current score, the number of foods Socrates has, the current level, etc.).

The *move()* method must return one of three different values when it returns at the end of each tick (all are defined in *GameConstants.h*):

```
GWSTATUS_PLAYER_DIED
GWSTATUS_CONTINUE_GAME
GWSTATUS_FINISHED_LEVEL
```

The first return value indicates that Socrates died during the current tick, and instructs our provided framework code to tell the user the bad news and restart the level if Socrates has more lives left (or end the game if he's out of lives). If your *move()* method returns this value and Socrates has more lives left, then our framework will prompt the player to continue the game, call your *cleanup()* method to destroy the level, call your *init()* method to re-initialize the Petri dish from scratch, and then begin calling your *move()* method over and over, once per tick, to let the user play the level again.

The second return value indicates that the tick completed without Socrates dying BUT Socrates has not yet completed the current level. Therefore the gameplay should continue normally for the time being. In this case, the framework will advance to the next tick and call your *move()* method again.

The final return value indicates that Socrates has completed the current level (that is, he successfully killed all of the bacteria, and all of the pits have emptied and disappeared). If your *move()* method returns this value, then the current level is over, and our framework will call your *cleanup()* method to destroy the level, our framework will then advance to the next level, then call your *init()* method to prepare that level for play, etc…

IMPORTANT NOTE: The skeleton code that we provide to you is hard-coded to return a `GWSTATUS_PLAYER_DIED` status value from our dummy version of the *move()* method. Unless you implement something that returns `GWSTATUS_CONTINUE_GAME` your game will not display any objects on the screen! So if the screen just immediately tells you that you lost a life once you start playing, you'll know why!

Here's pseudocode for how the move() method might be implemented:

```
int StudentWorld::move()
{
    // The term "actors" refers to all bacteria, Socrates, goodies,
    // pits, flames, spray, foods, etc.

      // Give each actor a chance to do something, incl. Socrates
    for each of the actors in the game world
    {
        if (the actor is still active/alive)
        {
            // tell that actor to do something (e.g. move)
          the actor->doSomething();

          if (Socrates during this tick)
                return GWSTATUS_PLAYER_DIED;

          if (Socrates completed the current level)
          {
                return GWSTATUS_FINISHED_LEVEL;
          }
        }
    }

      // Remove newly-dead actors after each tick
    Remove and delete dead game objects

      // Potentially add new actors to the game (e.g., goodies or fungi)
    Add new actors

      // Update the game status line
    Update display text   // update the score/lives/level text at screen top

      // the player hasn't completed the current level and hasn't died, so
      // continue playing the current level
    return GWSTATUS_CONTINUE_GAME;
}
```

Give Each Actor a Chance to Do Something

During each tick of the game each active actor must have an opportunity to do something (e.g., move around, shoot, etc.). Actors include Socrates, bacteria, projectiles like flames and spray (which are added to the game when Socrates attacks), food, goodies like restore health goodies, pits, and fungi.

Your *move()* method must iterate over every actor that's active in the game (i.e., held by your *StudentWorld* object) and ask it to do something by calling a member function in the actor's object named perhaps *doSomething()*. In each actor's *doSomething()* method, the object will have a chance to perform some activity based on the nature of the actor and its current state: e.g., a salmonella might move one pixel left, Socrates might shoot a flame or a spray, a flame may damage a nearby bacterium or pile of dirt, etc.

It is possible that one actor (e.g., a flame) may destroy another actor (e.g., a salmonella or pile of dirt) during the current tick. If an actor has died earlier in the current tick, then the dead actor must NOT have a chance to do something during the current tick (since it's dead). Also, other live actors processed during the tick must not interact with an actor

after it has died (e.g., Socrates should not get a bonus if he steps on top of a goodie that disappeared earlier during the same tick.).

To help you with testing, if you press the `f` key during the course of the game, our game controller will stop calling move() every tick; it will call move() only when you hit a key (except the `r` key). Freezing the activity this way gives you time to examine the screen, and stepping one move at a time when you're ready helps you see if your actors are moving properly. To resume regular gameplay, press the `r` key.

Remove Dead Actors After Each Tick

At the end of each tick, your *move()* method must determine which of your actors are no longer alive, remove them from your container of active actors, and use a C++ delete expression to free their objects (so you don't have a memory leak). So if, for example, a salmonella is killed by a spray and it dies, then it should be noted as dead, and at the end of the tick, its *pointer* should be removed from the StudentWorld's container of active objects, and the salmonella object should be deleted (using a C++ delete expression) to free up memory for future actors that will be introduced later in the game. Or, for example, after a flame has been fired and has travelled 32 pixels and is ready to dissipate, it must disappear from the screen and its object needs to be deleted as well. (Hint: Each of your actors could maintain a dead/alive status.)

Add New Goodies As Required After Each Tick

During each tick of the game, you may need to introduce a new goodie or fungus into the game for Socrates to pick up. Here are the rules for doing so on level L of the game:

1. ChanceFungus = *max*(510 – L * 10, **200**)
2. Generate a random number between [0, ChanceFungus)
3. If the random number is 0, then add a new fungus at a random angle/position around the circumference of the Petri dish, exactly VIEW_RADIUS pixels from the center of the Petri dish (which is at VIEW_WIDTH/2, VIEW_HEIGHT/2).
4. ChanceGoodie = *max*(510 – L * 10, **250**)
5. Generate a random number between [0, ChanceGoodie)
6. If the random number is 0, then add a new goodie at a random angle/position around the circumference of the Petri dish, exactly VIEW_RADIUS pixels from the center of the Petri dish (which is at VIEW_WIDTH/2, VIEW_HEIGHT/2).
   A. There is a 60% chance the goodie will be a restore health goodie.
   B. There is a 30% chance the goodie will be a flamethrower goodie.
   C. There is a 10% chance the goodie will be an extra-life goodie.

cleanUp() Details

When your *cleanUp()* method is called by our game framework, it means that Socrates lost a life (e.g., he was killed by a bacterium or fungi) or has completed the current level. In this case, every actor in the entire game (Socrates and every bacterium, goodie, projectile, food, fungus, pit, etc.) must be deleted and removed from the *StudentWorld's* container of active objects, resulting in an empty level. If the user has more lives left, our provided code will subsequently call your *init()* method to reload and repopulate the level with a new set of actors, and the level will then continue from scratch.

You must not call the *cleanUp()* method yourself when Socrates dies. Instead, this method will be called by our code when *init()* returns an appropriate status.

You Have to Create the Classes for All Actors

Kontagion has a number of different actors, including:

- Socrates
- Regular salmonella bacteria
- Aggressive salmonella bacteria
- E. coli bacteria
- Bacterial pits
- Flame projectiles
- Spray projectiles
- Dirt piles
- Food
- Restore health goodies
- Flamethrower goodies
- Extra life goodies
- Fungi

Each of these actor types can occupy your various levels and interact with other game actors within the visible screen view.

Now of course, many of your game actors will share things in common – for instance, every one of the actors in the game (salmonella, E. coli, Socrates, pits, flames, fungi, etc.) has (x, y) coordinates. Many game actors have the ability to perform an action (e.g., move, damage another actor) during each tick of the game. Many of them can potentially be attacked (e.g., Socrates, dirt piles, bacteria, goodies, fungus) and could "die" during a tick. Certain objects like flames, spray, fungi, and goodies "activate" when they come into contact with a proper target (e.g., goodies activate when they overlap with Socrates (and give his some special power), flames activate when they overlap with bacteria, goodie, fungus or a dirt pile (and destroy it), etc.

It is therefore your job to determine the commonalities between your different game classes and make sure to factor out common behaviors and traits and move these into appropriate base classes, rather than duplicate these items across your derived classes – this is in fact one of the tenets of object oriented programming.

Your grade on this project will depend upon your ability to intelligently create a set of classes that follow good object-oriented design principles. **Your classes must never duplicate code or a data member** – if you find yourself writing the same (or largely similar) code or duplicating member variables across multiple classes, then this is an indication that you should define a common base class and migrate this common functionality/data to the base class. Duplication of code is a so-called *code smell*, a weakness in a design that often leads to bugs, inconsistencies, code bloat, etc.

Hint: When you notice this specification repeating the same text nearly identically in the following sections (e.g., in the restore health goodie section and the flamethrower goodie section, or in the salmonella and E. coli sections) you must make sure to identify common behaviors and move these into proper base classes. NEVER duplicate behaviors (aka methods and member variables) across classes that can be moved into a base class!

You MUST derive all of your game objects directly or indirectly from a base class that we provide called *GraphObject*, e.g.:

```
class Actor: public GraphObject
{
public:
	…
};

class AggressiveSalmonella: public Actor
{
public:
	…
};

class Goodie: public Actor
{
public:
	…
};
```

GraphObject is a class that we have defined that helps hide the ugly logic required to graphically display your actors on the screen. If you don't derive your classes from our *GraphObject* base class, then you won't see anything displayed on the screen! ☺

The *GraphObject* class provides the following methods that you may use:

```
GraphObject(int imageID, double startX, double startY,
                                    int startDirection = 0, int depth = 0);
double getX() const;              // in pixels (0-255)
double getY() const;              // in pixels (0-255)
void moveTo(double x, double y); // in pixels (0-255)
```

```
// moveAngle() moves the actor the specified number of units in the
// specified direction.
void moveAngle(int angle, int units = 1);
// getPositionInThisDirection() returns a new (x, y) location in the
// specified direction and distance, based on the passed-in angle and the
// GraphObject's current (x, y) location.
void getPositionInThisDirection(int angle, int units,
                                                double& dx, double& dy);
int getDirection() const;  // in degrees (0-359)
void setDirection(int d);  // in degrees (0-359)
```

You may use any of these member functions in your derived classes, but you **must not** use any other member functions found inside of *GraphObject* in your other classes (even if they are public in our class). You **must not** redefine any of these methods in your derived classes **unless** they are marked as virtual in our base class.

GraphObject(int imageID,
int startX, *// column first*
int startY, *// then row!*
int startDirection,
int depth)

is the constructor for a new *GraphObject*. When you construct a new *GraphObject*, you must specify an image ID that indicates how the *GraphObject* should be displayed on screen (e.g., as a salmonella, fungus, Socrates, a pit, a flame, etc.). You must also specify the initial (x, y) location of the object. The x value may range from 0 to VIEW_WIDTH-1 inclusive, and the y value may range from 0 to VIEW_HEIGHT-1 inclusive (these constants are defined in our provided header file *GameConstants.h*). Notice that you pass the coordinates as x, y (i.e., column, row starting from bottom left, and ***not*** row, column). You may also specify the initial direction an object is facing as an angle between 0-359 degrees. Finally, you must specify the depth of the object. An object of depth 0 is in the foreground, whereas objects with increasing depths are drawn further in the background. Thus an object of depth zero always covers object with a depth of one or greater, and an object with a depth of one always covers objects of depth two or greater, etc. In Kontagion, all bacteria and Socrates are at depth 0, and all other game actors are at depth 1, ensuring our active characters are in the foreground.

One of the following IDs from *GameConstants.h* must be passed in as the *imageID* value:

```
IID_PLAYER (for Socrates)
IID_SALMONELLA
IID_ECOLI
IID_FOOD
IID_SPRAY
IID_FLAME
IID_PIT
IID_DIRT
IID_FLAME_THROWER_GOODIE
IID_RESTORE_HEALTH_GOODIE
IID_EXTRA_LIFE_GOODIE
IID_FUNGUS
```

If you derive your game objects from our *GraphObject* class, they will be displayed on screen automatically by our framework (e.g., a salmonella image will be drawn to the screen at the *GraphObject's* specified x,y coordinates if the object's Image ID is IID_ SALMONELLA).

The classes you write MUST NOT store an imageId value or any value somehow related/derived from the imageID value in any way or you will get a Zero on this project. Only our GraphObject class may store the imageId value.

getX() and *getY()* are used to determine a *GraphObject's* current location in the level. Since each *GraphObject* maintains its own (x, y) location, this means that your derived classes **must NOT** also have x or y member variables, but instead use these functions and *moveTo()* from the *GraphObject* base class.

moveTo(double x, double y) is used to update the location of a *GraphObject* within the level. For example, if a salmonella's movement logic dictates that it should move one pixel to the left, you could do the following:

```
moveTo(getX()-1, y);       // move one pixel to the left
```

moveAngle(int angle, int units = 1) is used to update the location of a *GraphObject* within the level. For example, if an actor's movement logic dictates that it should move 5 pixels in the direction it is facing, you could do the following:

```
moveAngle(getDirection(), 5);  // move 5 pixels in current direction
```

You ***must*** use the *moveTo()* or *moveAngle()* methods to adjust the location of a game object if you want that object to be properly animated. As with the *GraphObject* constructor, note that the order of the parameters to *moveTo* is x,y (col, row) and NOT y, x (row,col).

getDirection() is used to determine the direction a *GraphObject* is facing, and returns a value of *0-359*.

setDirection(int d) is used to change the direction a *GraphObject* is facing and takes a value between 0 and 359. For example, you could use this method and *getDirection()* to adjust the direction of a bacterium when it decides to move in a new direction. Note that an actor can be facing in one direction, but move in a totally different direction using *moveTo()* or *moveAngle()*. The angle of movement and the angle the actor is facing are allowed to be totally different.

Socrates

Here are the requirements you must meet when implementing the Socrates class.

What a Socrates Object Must Do When It Is Created

When it is first created:

1. A Socrates object must have an image ID of IID_PLAYER.
2. A Socrates object starts out alive.
3. A Socrates object has a direction (that it faces) of 0 degrees (i.e., to the right).
4. A Socrates object has a starting positional angle of 180 degrees relative ti the center of the Petri dish, meaning that it starts at the location X=0, Y=128, on the left side of the Petri dish. A positional angle of 0 degrees would put Socrates at the far right side of the petri dish at X=256, Y=128, and an angle of 90 degrees would put Socrates at X=128, Y=256, etc.
5. A Socrates object has a depth of 0.
6. A Socrates object starts with 20 spray charges and 5 flamethrower charges.
7. A Socrates object starts out with 100 hit points (health).

What a Socrates Object Must Do During a Tick

Socrates must be given an opportunity to do something during every tick (in his *doSomething()* method). When given an opportunity to do something, Socrates must do the following:

1. Socrates must check to see if he is currently alive (i.e., his hit points are above zero). If not, then Socrates's *doSomething()* method must return immediately – none of the following steps should be performed.
2. The *doSomething()* method must check to see if the player pressed a key (the section below shows how to check this). If the player pressed a key:
    a. If the player pressed the Space key and Socrates has at least one spray charge left, then Socrates will:
        i. Add a spray object SPRITE_WIDTH pixels directly in front of himself (in the same direction Socrates is facing, which is always toward the center of the Petri dish).
        ii. Socrates's spray count must decrease by 1.
        iii. Socrates must play the SOUND_PLAYER_SPRAY sound effect (see the *StudentWorld* section of this document for details on how to play a sound).
    b. If the player pressed the Enter key and Socrates has at least one flamethrower charge left, then Socrates will:
        i. Add 16 flame objects exactly SPRITE_WIDTH pixels away from himself, starting directly in front of Socrates and in 22 degree increments around Socrates (making a complete circle of flames around Socrates). Each flame object must face the same

direction/angle that it was created around Socrates. e.g., a flame object created at 44 degrees from the horizontal should have a direction it faces of 44 degrees.
        ii. Socrates's flamethrower count must decrease by 1.
        iii. Socrates must play the SOUND_PLAYER_FIRE sound effect (see the *StudentWorld* section of this document for details on how to play a sound).
    c. If the user asks to move clockwise or counterclockwise by pressing a directional key:
        i. Adjust Socrates's positional angle elative to the center of the Petri dish either clockwise or counterclockwise by 5 degrees.
        ii. Adjust Socrates's (x, y) coordinates around the perimeter of the Petri dish based on his new positional angle.
        iii. Set the direction that Socrates is facing to (180 degrees + his positional angle) % 360, causing him to always face toward the center of the Petri dish.
3. If the player did NOT press a key, then:
    a. The *doSomething()* method must check to see if Socrates has fewer than the maximum number of spray charges (20 charges), and if he has less than the maximum, increase his available number of sprays by 1 spray. Socrates must wait one or more ticks without spraying to replenish his spray charges.

<u>What a Socrates Object Must Do In Other Circumstances</u>

- Socrates can be damaged. If a fungus object or any type of bacteria object overlaps with Socrates, it will cause a certain amount of damage, reducing Socrates' hit points.
    - Socrates must lower his hit points based on the amount of damage specified by the other actor.
    - If Socrates is damaged (but not killed), he must play a SOUND_PLAYER_HURT sound effect.
    - Otherwise, if Socrates hit points reach zero (or below), the Socrates object must:
        - Immediately has his status set to dead.
        - He must play a SOUND_PLAYER_DIE sound effect.
        - (The *StudentWorld* class should later detect his death and the current level ends)
- Socrates does NOT block other objects from moving near or onto his location.

Getting Input From the User

Since Kontagion is a *real-time* game, you can't use the typical *getline* or *cin* approach to get a user's key press within Socrates's *doSomething()* method— that would stop your program and wait for the user to type something and then hit the Enter key. This would make the game awkward to play, requiring the user to hit a directional key then hit Enter, then hit a directional key, then hit Enter, etc. Instead of this approach, you will use a function called *getKey()* that we provide in our *GameWorld* class (from which your *StudentWorld* class is derived) to get input from the player[1]. This function rapidly checks to see if the user has hit a key. If so, the function returns true and the int variable passed to it is set to the code for the key. Otherwise, the function immediately returns false, meaning that no key was hit. This function could be used as follows:

```
void Socrates::doSomething()
{
    ...
    int ch;
    if (getAPointerToMyWorld()->getKey(ch))
    {
        // user hit a key during this tick!
        switch (ch)
        {
          case KEY_PRESS_LEFT:
            ... move Socrates counterclockwise ...;
            break;
          case KEY_PRESS_RIGHT:
            ... move Socrates clockwise...;
            break;
          case KEY_PRESS_SPACE:
            ... add spray in front of Socrates...;
            break;

          // etc…
        }
    }
    ...
}
```

Dirt Pile

Dirt piles don't really do much. They just sit there and vegetate like USC students. Here are the requirements you must meet when implementing a dirt pile class.

What a Dirt Pile Must Do When It Is Created

When it is first created:

1. A dirt pile object must have an image ID of IID_DIRT.

[1] Hint: Since your Socrates class will need to access the *getKey()* method in the *GameWorld* class (which is the base class for your *StudentWorld* class), your Socrates class (or more likely, one of its base classes) will need a way to obtain a pointer to the *StudentWorld* object it's playing in. If you look at our code example, you'll see how Socrates's *doSomething()* method first gets a pointer to its world via a call to *getAPointerToMyWorld()* (a method in one of its base classes that returns a pointer to a *StudentWorld*), and then uses this pointer to call the *getKey()* method.

2. A dirt pile object must have its (x, y) location specified for it as detailed in the *StudentWorld::init()* section of this document. *Hint: Your StudentWorld class can pass in this (x, y) location when constructing a dirt pile object.*
3. A dirt pile object has a direction of 0 degrees.
4. A dirt pile object has a depth of 1.
5. A dirt object starts out in the alive state.
6. A dirt object has no hit points; it is either fully alive or fully dead.

What a Dirt Pile Must Do During a Tick

A dirt pile must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the dirt pile must do nothing. C'mon – it's dirt. What could it do?

What a Dirt Pile Must Do In Other Circumstances

- A Dirt pile can be damaged. A single spray or flame that hits the dirt pile will destroy it (and cause the spray/flame to dissipate once it has hit the pile).
- A Dirt pile blocks the movement of all bacteria (the Euclidean distance of the bacterium and of the dirt pile must not be less than SPRITE_WIDTH/2 pixels from each other).
- A Dirt pile blocks the movement of all spray and flames.

Food

Food doesn't really do much. It just sits there waiting for bacteria to eat it. Here are the requirements you must meet when implementing a food class.

What Food Must Do When It Is Created

When it is first created:

1. A food object must have an image ID of IID_FOOD.
2. A food object must have its (x, y) location specified for it as detailed in the *StudentWorld::init()* section of this document. *Hint: Your StudentWorld class can pass in this (x, y) location when constructing a food object.*
3. A food object has a direction of 90 degrees.
4. A food object has a depth of 1.
5. A food object starts out in the alive state.
6. A food object has no hit points; it is either fully alive or fully dead.

What Food Must Do During a Tick

A food object must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the

food object must do nothing. You might think it should decay over time, but it has lots of preservatives, so its half-life is probably a few thousand years.

What Food Must Do In Other Circumstances

- Food cannot be damaged by spray or flames.
- Food does not block the movement of bacteria.
- Food does not block the movement of spray or flames.

Flame

You must create a class to represent a flame which Socrates can use to fry bacteria, goodies, fungi and dirt piles. Flame objects are produced from Socrates when he fires his flamethrower. Here are the requirements you must meet when implementing a flame class.

What a Flame Must Do When It Is Created

When it is first created:

1. A flame object must have an image ID of IID_FLAME.
2. The starting location of a flame must be specified during construction.
3. The starting direction of a flame must be specified during construction.
4. All flames start out with a maximum travel distance of 32 pixels
5. All flames have a depth of 1.
6. All flames start in the alive state.
7. A flame object has no hit points; it is either fully alive or fully dead.

What a Flame Must Do During a Tick

A flame must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the flame must:

1. Check to see if it's still alive. If not, it must immediately return. Otherwise…
2. Check to see if it overlaps with a *damageable object* (e.g., a bacterium of any type, goodie, fungus, or a dirt pile). To check for overlap, see if the flame has a Euclidean distance of <= SPRITE_WIDTH between itself and a target object's (x, y) coordinate. If so, the flame must:
    a. Tell a single impacted object that it has been damaged with 5 hit points of damage (if the flame overlaps with multiple objects at the same time, it must damage only one of them – which one is left to **you** to decide. Don't ask us which object you should damage.).
    b. Immediately set its status to dead, so the flame goes away immediately.
    c. Return immediately.

3. Otherwise, the flame will move forward in its current direction by SPRITE_WIDTH pixels.
4. If the flame has moved a total of 32 pixels after step 3, then it immediately sets its status to dead (it dissipates) and can cause no more damage.

What a Flame Must Do In Other Circumstances

- A Flame cannot be damaged (they don't have hit points).
- A Flame does not block any other objects from moving near or onto it.

Disinfectant Spray

You must create a class to represent a disinfectant spray which Socrates can use to kill bacteria and dissolve dirt piles. Spray objects are produced from Socrates's spray bottle. Here are the requirements you must meet when implementing the spray class.

What a Spray Must Do When It Is Created

When it is first created:

1. A spray object must have an image ID of IID_SPRAY.
2. The starting location of a spray must be specified during construction.
3. The starting direction of a spray must be specified during construction.
4. All sprays start out with a maximum travel distance of 112 pixels
5. All sprays have a depth of 1.
6. All sprays start in the alive state.
7. A spray object has no hit points; it is either fully alive or fully dead.

What a Spray Must Do During a Tick

A spray must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the spray must:

1. Check to see if it's still alive. If not, it must immediately return. Otherwise…
2. Check to see if it overlaps with a *damageable object* (e.g., a bacterium of any type, goodie, fungus, or a dirt pile). To check for overlap, see if the spray has a Euclidean distance of <= SPRITE_WIDTH between itself and the target object. If so, the spray must:
    a. Tell a single impacted object that it has been damaged with 2 hit points of damage (if the spray overlaps with multiple objects at the same time, it must damage only one of them – which one is left to you to decide. Don't ask us which object should be chosen.).
    b. Immediately set its status to dead, so the spray goes away immediately.
    c. Return immediately.

3. Otherwise, the spray will move forward in its current direction by SPRITE_WIDTH pixels.
4. If the spray has moved a total of 112 pixels after step 3, then it immediately sets its status to dead (it dissipates) and can cause no more damage.

What a Spray Must Do In Other Circumstances

- A Spray cannot be damaged (they don't have hit points).
- A Spray does not block any other objects from moving near or onto it.

Restore Health Goodie

You must create a class to represent a restore health goodie that Socrates can pick up. When Socrates overlaps with (picks up) this goodie, it restores him to his full 100 hit points. Here are the requirements you must meet when implementing the restore health goodie class.

What a Restore Health Goodie Must Do When It Is Created

When it is first created:

1. A restore health goodie object must have an image ID of IID_RESTORE_HEALTH_GOODIE.
2. The starting location of a restore health goodie must be specified during construction.
3. A restore health goodie has a direction of 0 degrees.
4. A restore health goodie has a depth of 1.
5. A restore health goodie starts in an "alive" state.
6. A restore health goodie has a lifetime of *max*(*randInt*(0, 300 – 10 * L - 1), 50) ticks on level L before it disappears.
7. A restore health goodie has no hit points; it is either fully alive or fully dead.

What a Restore Health Goodie Must Do During a Tick

Each time a restore health goodie is asked to do something (during a tick):

1. The restore health goodie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed. Otherwise…
2. The restore health goodie must determine if it overlaps[2] with Socrates. If so, then the restore health goodie must:
    a. Inform the *StudentWorld* object that the user is to receive 250 points.

[2] Same definition of overlap as flames, spray, etc. If the Euclidean distance between the goodie and Socrates is <= SPRITE_WIDTH, then the two overlap.

    b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    c. Play a sound effect to indicate that Socrates picked up the goodie: SOUND_GOT_GOODIE.
    d. Tell the Socrates object to restore its health to full.
    e. Return immediately…
3. The restore health goodie must check to see if its lifetime has expired, and if so, set its state to dead so it disappears from the game.

What a Restore Health Goodie Must Do In Other Circumstances

- A Restore health goodie can be damaged by flames and sprays. When damaged it will set its status to dead and disappear from the game.
- A Restore health goodie does not block other objects from moving near or onto it.

Flamethrower Goodie

You must create a class to represent a flamethrower goodie that Socrates can pick up. When Socrates overlaps with (picks up) this goodie, it gives him **five** additional flamethrower charges. Here are the requirements you must meet when implementing the flamethrower goodie class.

What a Flamethrower Goodie Must Do When It Is Created

When it is first created:

1. A flamethrower goodie object must have an image ID of IID_FLAME_THROWER_GOODIE.
2. The starting location of a flamethrower goodie must be specified during construction.
3. A flamethrower goodie has a direction of 0 degrees.
4. A flamethrower goodie has a depth of 1.
5. A flamethrower goodie starts in an "alive" state.
6. A flamethrower goodie has a lifetime of *max*(*randInt*(0, 300 – 10 * L - 1), 50) ticks on level L before it disappears.
7. A flamethrower goodie has no hit points; it is either fully alive or fully dead.

What a Flamethrower Goodie Must Do During a Tick

Each time a flamethrower goodie is asked to do something (during a tick):

1. The flamethrower goodie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed. Otherwise…

2. The flamethrower goodie must determine if it overlaps[3] with Socrates. If so, then the flamethrower goodie must:
   a. Inform the *StudentWorld* object that the user is to receive 300 points.
   b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   c. Play a sound effect to indicate that Socrates picked up the goodie: SOUND_GOT_GOODIE.
   d. Tell the Socrates object to add 5 flamethrower charges to its arsenal.
   e. Return immediately…
3. The flamethrower goodie must check to see if its lifetime has expired, and if so, set its state to dead so it disappears from the game.

What a Flamethrower Goodie Must Do In Other Circumstances

- A Flamethrower goodie can be damaged by flames and sprays. When damaged it will set its status to dead and disappear from the game.
- A Flamethrower goodie does not block other objects from moving near or onto it.

Extra Life Goodie

You must create a class to represent an extra life goodie that Socrates can pick up. When Socrates overlaps with (picks up) this goodie, it increases the number of lives he has. Here are the requirements you must meet when implementing the extra life goodie class.

What an Extra life Goodie Must Do When It Is Created

When it is first created:

1. An extra life goodie object must have an image ID of IID_EXTRA_LIFE_GOODIE.
2. The starting location of an extra life goodie must be specified during construction.
3. An extra life goodie has a direction of 0 degrees.
4. An extra life goodie has a depth of 1.
5. An extra life goodie starts in an "alive" state.
6. An extra life goodie has a lifetime of *max*(*randInt*(0, 300 – 10 * L - 1), 50) ticks on level L before it disappears.
7. An extra life goodie has no hit points; it is either fully alive or fully dead.

What an Extra life Goodie Must Do During a Tick

Each time an extra life goodie is asked to do something (during a tick):

[3] Same definition of overlap as flames, spray, etc. If the Euclidean distance between the goodie and Socrates is less than SPRITE_WIDTH, then the two overlap.

1. The extra life goodie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed. Otherwise…
2. The extra life goodie must determine if it overlaps[4] with Socrates. If so, then the extra life goodie must:
   a. Inform the *StudentWorld* object that the user is to receive 500 points.
   b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   c. Play a sound effect to indicate that Socrates picked up the goodie: SOUND_GOT_GOODIE.
   d. Tell the StudentWorld class that the player has earned an extra life by calling *StudentWorld::incLives()* method (this method is inherited by *StudentWorld* from *GameWorld*).
   e. Return immediately…
3. The extra life goodie must check to see if its lifetime has expired, and if so, set its state to dead so it disappears from the game.

What an Extra Life Goodie Must Do In Other Circumstances

- An Extra life goodie can be damaged by flames and sprays. When damaged it will set its status to dead and disappear from the game.
- An Extra life goodie does not block other objects from moving near or onto it.

Fungus

You must create a class to represent a fungus that will injure Socrates. When Socrates overlaps with (picks up) this fungus, it reduces his hit points by 20 points, potentially killing Socrates. Here are the requirements you must meet when implementing the fungus class.

What a Fungus Must Do When It Is Created

When it is first created:

1. A fungus object must have an image ID of IID_FUNGUS.
2. The starting location of a fungus must be specified during construction.
3. A fungus has a direction of 0 degrees.
4. A fungus has a depth of 1.
5. A fungus starts in an "alive" state.
6. A fungus has a lifetime of *max*(*randInt*(0, 300 – 10 * L - 1), 50) ticks on level L before it disappears.
7. A fungus has no hit points; it is either fully alive or fully dead.

[4] Same definition of overlap as flames, spray, etc. If the Euclidean distance between the goodie and Socrates is less than SPRITE_WIDTH, then the two overlap.

What a Fungus Must Do During a Tick

Each time a fungus is asked to do something (during a tick):

1. The fungus must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed. Otherwise…
2. The fungus must determine if it overlaps[5] with Socrates. If so, then the fungus must:
   a. Inform the *StudentWorld* object that the user is to receive −50 points.
   b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   c. Tell the Socrates object that it has received 20 points of damage.
   d. Return immediately…
3. The fungus must check to see if its lifetime has expired, and if so, set its state to dead so it disappears from the game.

What a Fungus Must Do In Other Circumstances

- A Fungus can be damaged by flames and sprays. When damaged it will set its status to dead and disappear from the game.
- A Fungus does not block other objects from moving near or onto it.

Pit (aka Bacteria Generator)

You must create a class to represent a pit that produces bacteria in the Petri dish. Here are the requirements you must meet when implementing the pit class.

What a Pit Must Do When It Is Created

When it is first created:

1. A pit object must have an image ID of IID_PIT.
2. The starting location of a pit must be specified during construction.
3. A pit has a direction of 0 degrees.
4. A pit has a depth of 1.
5. A pit starts with an initial inventory of 5 regular salmonella, 3 aggressive salmonella, and 2 E. coli.
6. A pit starts out in an “alive” state.
7. A pit has no hit points; it is either fully alive or fully dead.

[5] Same definition of overlap as flames, spray, etc. If the Euclidean distance between the goodie and Socrates is less than SPRITE_WIDTH, then the two overlap.

What a Pit Must Do During a Tick

Each time a pit is asked to do something (during a tick):

1. The pit must determine whether or not it has emitted all of its bacteria (all 5 regular salmonella, 3 aggressive salmonella, and 2 E. coli have been generated and added to the Petri dish). If it has emitted all of its bacteria, it must:
    a. Inform the *StudentWorld* object that it has emitted all of its bacteria[6].
    b. Set its status to dead so it disappears from the game at the end of the current tick.
2. Otherwise, there is a 1 in 50 chance that during a tick, a given pit will emit a bacterium. Assuming the current tick was chosen to emit bacteria into the Petri dish:
    a. Considering only the types of bacteria that is has not yet run out of, the pit must randomly select one of those types (each type being equally likely to be chosen).
    b. It must create a new bacterium object of the chosen type (regular salmonella, aggressive salmonella or E. coli) with an (x ,y) coordinate that is the same as the pit's (x ,y) coordinate.
    c. It must add that new bacterium to the *StudentWorld* object.
    d. It must decrement the count of bacteria of the chosen type that it has left in its inventory.
    e. It (or some other object) must play a sound effect to indicate that the bacterium was just born: SOUND_BACTERIUM_BORN.

What a Pit Must Do In Other Circumstances

- A Pit cannot be damaged by sprays or flames.
- A Pit does not block the movement of sprays or flames.
- A Pit does not block other objects from moving near or onto it.

Regular Salmonella

You must create a class to represent a regular salmonella bacterium. Here are the requirements you must meet when implementing the regular salmonella class.

What a Regular Salmonella Must Do When It Is Created

When it is first created:

[6] This is optional. All that is required is that *StudentWorld* can determine when all pits have emitted all of their bacteria (and all of the bacteria have been killed by Socrates). This way the *StudentWorld* class can determine when the level has been completed.

1. A regular salmonella object must have an image ID of IID_SALMONELLA.
2. A regular salmonella must always start at the proper location as passed into its constructor.
3. A regular salmonella has a direction of 90 degrees.
4. A regular salmonella has a depth of 0.
5. A regular salmonella starts with a *movement plan distance* of 0, meaning it has no plan to move in its current direction.
6. A regular salmonella starts out in an "alive" state.
7. A regular salmonella starts with 4 hit points.

What a Regular Salmonella Must Do During a Tick

Each time a regular salmonella is asked to do something (during a tick):

1. The regular salmonella must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. The regular salmonella must check to see if it overlaps[7] with Socrates. If so it will:
    a. Tell Socrates that he has received 1 point of damage.
    b. Then skip to step 5.
3. The regular salmonella will see if it's eaten a total of 3 food since it last divided or was born. If so:
    a. It will compute a *newx* coordinate, equal to its own *x* coordinate:
        i. Plus SPRITE_WIDTH/2 if its *x* coordinate is < VIEW_WIDTH/2
        ii. Minus SPRITE_WIDTH/2 if its x coordinate is > VIEW_WIDTH/2
        iii. If its x coordinate is equal to VIEW_WIDTH/2
    b. It will compute a *newy* coordinate, equal to its own *y* coordinate:
        i. Plus SPRITE_WIDTH/2 if its *y* coordinate is < VIEW_HEIGHT/2
        ii. Minus SPRITE_WIDTH/2 if its y coordinate is > VIEW_ HEIGHT /2
        iii. If its y coordinate is equal to VIEW_HEIGHT/2
    c. It will add a new regular salmonella object at the specified (newx, newy) coordinate.
    d. It will reset its food-eaten count back to zero (so it must now eat three food objects before it can divide again).
    e. Then skip to step 5.
4. The regular salmonella will check to see if it overlaps[8] with one or more food objects. If so, it will:
    a. Increase its food eaten count by 1.
    b. Select one of the food objects it overlaps with and tell it to set its state to dead so the food disappears. If it overlaps with more than one food

[7] Same definition of overlap as used with flames, spray, etc.
[8] Same definition of overlap as used with flames, spray, etc.

objects, then it must pick one (we don't care which one; just pick one) to eat.

5. The regular salmonella will check to see if it has a *movement plan distance* of greater than zero. If so this means it wants to continue moving the same direction it was previously moving in. It will do the following:
    a. Decrement its *movement plan distance* by one.
    b. It will see if it can move forward by 3 pixels in the direction it is currently facing. The following things will block the regular salmonella from moving forward:
        i. Moving forward 3 pixels would cause the regular salmonella to overlap with a dirt pile, **where in this case, the definition of overlap means that the Euclidean distance from the regular salmonella's proposed new (x, y) location to the dirt pile is <= SPRITE_WIDTH/2 pixels.**
        ii. Moving forward 3 pixels would cause the regular salmonella to move outside of the Petri dish, that is, its distance from the center of the screen (VIEW_WIDTH/2, VIEW_HEIGHT/2) is greater than or equal to VIEW_RADIUS.
    c. If the regular salmonella **can** move forward 3 pixels to the new position, it (or some base class it's derived from) must do so using either the *GraphObject moveTo()* or *moveAngle()* method.
    d. Otherwise, if the regular salmonella is blocked from moving forward 3 pixels, it will:
        i. Pick a random direction to move in, from 0 to 359 degrees, and set its direction to that new direction.
        ii. Reset its *movement plan distance* to 10, so it will try to move 10 ticks in this new direction.
    e. The regular salmonella will immediately return.
6. Otherwise, the regular salmonella will get the directional angle to the closest food within 128 pixels of itself on the Petri dish.
    a. If no food can be found within 128 pixels, then the regular salmonella will:
        i. Pick a random direction to move in, from 0 to 359 degrees, and set its direction to that direction.
        ii. Reset its *movement plan distance* to 10, so it will try to move 10 ticks in this new direction.
        iii. Immediately return.
    b. If food can be found within 128 pixels, the regular salmonella will try to move toward that food. If moving forward 3 pixels would cause the regular salmonella to overlap[9] with a dirt pile, then it will:
        i. Pick a random direction to move in, from 0 to 359 degrees, and set its direction to that new direction.
        ii. Reset its *movement plan distance* to 10, so it will try to move 10 ticks in this new direction.
        iii. Immediately return.

[9] Again, overlap in this case means moving to within SPRITE_WIDTH/2 pixels of the dirt pile.

What a Regular Salmonella Must Do In Other Circumstances

- Regular salmonella can be damaged by spray and flames. If a spray or flame hits a regular salmonella, then the regular salmonella must:
    - Reduce its hit points by the specified amount (as dictated by the spray/flame).
    - If the regular salmonella is still alive then it must play a sound of SOUND_SALMONELLA_HURT.
    - Otherwise if a spray or flame reduces the regular salmonella's hit points to 0 or below (killing it), the regular salmonella must:
        - Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
        - Play a sound effect to indicate that the regular salmonella died: SOUND_SALMONELLA_DIE.
        - Increase the player's score by 100 points by using the *StudentWorld::increaseScore()* method.
        - There is a 50% chance the regular salmonella will turn into food before it dies. If so, the regular salmonella will introduce a new food object in its current (x,y) coordinate by adding it to the *StudentWorld* object.
- Regular salmonella do not block other objects from moving near or onto them.

Aggressive Salmonella

You must create a class to represent an aggressive salmonella bacterium. Here are the requirements you must meet when implementing the aggressive salmonella class.

What an Aggressive Salmonella Must Do When It Is Created

When it is first created:

1. An aggressive salmonella object must have an image ID of IID_SALMONELLA.
2. An aggressive salmonella must always start at the proper location as passed into its constructor.
3. An aggressive salmonella has a direction of 90 degrees.
4. An aggressive salmonella has a depth of 0.
5. An aggressive salmonella starts with a *movement plan distance* of 0, meaning it has no plan to move in its current direction.
6. An aggressive salmonella starts out in an "alive" state.
7. An aggressive salmonella starts with 10 hit points.

What an Aggressive Salmonella Must Do During a Tick

Each time an aggressive salmonella is asked to do something (during a tick):

1. The aggressive salmonella must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. The aggressive salmonella must check to see if it's less than or equal to 72 pixels away from Socrates. If so:
    a. It will try to move forward 3 pixels in the direction toward Socrates.
    b. If it gets blocked by dirt, then it will not move (it will get stuck behind the dirt, and **not** try to find a new movement direction).
    c. Next it will perform steps 3-5 from below, and then immediately return. It **must NOT** perform steps 6 and beyond in this case.
3. The aggressive salmonella must check to see if it overlaps[10] with Socrates. If so it will:
    a. Tell Socrates that he has received 2 points of damage.
    b. Then skip to step 6 (unless instructed not to by step 2c, in which case just return immediately).
4. The aggressive salmonella will see if it's eaten a total of 3 food since it last divided or was born. If so:
    a. It will compute a *newx* coordinate, equal to its own *x* coordinate:
        i. Plus SPRITE_WIDTH/2 if its *x* coordinate is < VIEW_WIDTH/2
        ii. Minus SPRITE_WIDTH/2 if its x coordinate is >VIEW_WIDTH/2
        iii. If its x coordinate is equal to VIEW_WIDTH/2
    b. It will compute a *newy* coordinate, equal to its own *y* coordinate:
        i. Plus SPRITE_WIDTH/2 if its *y* coordinate is < VIEW_HEIGHT/2
        ii. Minus SPRITE_WIDTH/2 if its y coordinate is > VIEW_ HEIGHT /2
        iii. If its y coordinate is equal to VIEW_HEIGHT/2
    c. It will add a new aggressive salmonella object at the specified (newx, newy) coordinate.
    d. It will reset its food-eaten count back to zero (so it must now eat three food objects before it can divide again).
    e. Then skip to step 6 (unless instructed not to by step 2c, in which case just return immediately).
5. The aggressive salmonella will check to see if it overlaps[11] with one or more food objects. If so, it will:
    a. Increase its food eaten count by 1.
    b. Select one of the food objects it overlaps with and tell it to set its state to dead so the food disappears. If it overlaps with more than one food objects, then it must pick one (we don't care which one) to eat.
6. The aggressive salmonella will check to see if it has a *movement plan distance* of greater than zero. If so, this means it wants to continue moving the same direction it was previously moving in. It will do the following:
    a. Decrement its *movement plan distance* by one.

[10] Same definition of overlap as used with flames, spray, etc.
[11] Same definition of overlap as used with flames, spray, etc.

b. It will see if it can move forward 3 pixels in the direction it is currently facing. The following things will block the aggressive salmonella from moving forward:
    i. Moving forward 3 pixels would cause the aggressive salmonella to overlap with a dirt pile, **where in this case, the definition of overlap means that the Euclidean distance from the aggressive salmonella's proposed new (x, y) location to the dirt pile is <= SPRITE_WIDTH/2 pixels.**
    ii. Moving forward 3 pixels would cause the aggressive salmonella to move outside of the Petri dish, that is, its distance from the center of the screen (VIEW_WIDTH/2, VIEW_HEIGHT/2) is greater than or equal to VIEW_RADIUS.
c. If the aggressive salmonella **can** move forward 3 pixels to the new position, it (or some base class it's derived from) must do so using the *GraphObject moveTo()* or *moveAngle()* method.
d. Otherwise, if the aggressive salmonella is blocked from moving forward 3 pixels, it will:
    i. Pick a random direction to move in, from 0 to 359 degrees, and set its direction to that new direction.
    ii. Reset its *movement plan distance* to 10, so it will try to move 10 ticks in this new direction.
e. The aggressive salmonella will immediately return.

7. Otherwise, the aggressive salmonella will get the directional angle to the closest food within 128 pixels of itself on the Petri dish.
    a. If no food can be found within 128 pixels, then the aggressive salmonella will:
        i. Pick a random direction to move in, from 0 to 359 degrees, and set its direction to that direction.
        ii. Reset its *movement plan distance* to 10, so it will try to move 10 ticks in this new direction.
        iii. Immediately return.
    b. If food can be found within 128 pixels, the aggressive salmonella will try to move toward that food. If moving forward 3 pixels would cause the aggressive salmonella to overlap[12] with a dirt pile, then it will:
        i. Pick a random direction to move in, from 0 to 359 degrees, and set its direction to that new direction.
        ii. Reset its *movement plan distance* to 10, so it will try to move 10 ticks in this new direction.
        iii. Immediately return.

<u>What Aggressive Salmonella Must Do In Other Circumstances</u>

- Aggressive salmonella can be damaged by spray and flames. If a spray or flame hits an aggressive salmonella, then the aggressive salmonella must:

[12] Again, overlap in this case means moving to within SPRITE_WIDTH/2 pixels of the dirt pile.

    - Reduce its hit points by the specified amount (as dictated by the spray/flame).
    - If the aggressive salmonella is still alive then it must play a sound of SOUND_SALMONELLA_HURT.
    - Otherwise if a spray or flame reduces the aggressive salmonella's hit points to 0 or below (killing it), the aggressive salmonella must:
        - Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
        - Play a sound effect to indicate that the aggressive salmonella died: SOUND_SALMONELLA_DIE.
        - Increase the player's score by 100 points by using the *StudentWorld::increaseScore()* method.
        - There is a 50% chance the aggressive salmonella will turn into food before it dies. If so, the aggressive salmonella will introduce a new food object in its current (x,y) coordinate by adding it to the *StudentWorld* object.
- Aggressive salmonella do not block other objects from moving near or onto them.

E. coli

You must create a class to represent an E. coli bacterium. Here are the requirements you must meet when implementing the E. coli class.

What an E. coli Must Do When It Is Created

When it is first created:

1. An E. coli object must have an image ID of IID_ECOLI.
2. An E. coli must always start at the proper location as passed into its constructor.
3. An E. coli has a direction of 90 degrees.
4. An E. coli has a depth of 0.
5. An E. coli starts with a *movement plan distance* of 0, meaning it has no plan to move in its current direction.
6. An E. coli starts out in an "alive" state.
7. An E. coli starts with 5 hit points.

What an E. coli Must Do During a Tick

Each time an E. coli is asked to do something (during a tick):

1. The E. coli must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. The E. coli must check to see if it overlaps[13] with Socrates. If so it will:

[13] Same definition of overlap as used with flames, spray, etc.

    a. Tell Socrates that he has received 4 points of damage.
    b. Then skip to step 5.
3. The E. coli will see if it's eaten a total of 3 food since it last divided or was born. If so:
    a. It will compute a *newx* coordinate, equal to its own *x* coordinate:
        i. Plus SPRITE_WIDTH/2 if its *x* coordinate is < VIEW_WIDTH/2
        ii. Minus SPRITE_WIDTH/2 if its x coordinate is > VIEW_WIDTH/2
        iii. If its x coordinate is equal to VIEW_WIDTH/2
    b. It will compute a *newy* coordinate, equal to its own *y* coordinate:
        i. Plus SPRITE_WIDTH/2 if its *y* coordinate is < VIEW_HEIGHT/2
        ii. Minus SPRITE_WIDTH/2 if its y coordinate is > VIEW_ HEIGHT /2
        iii. If its y coordinate is equal to VIEW_HEIGHT/2
    c. It will add a new E. coli object at the specified (newx, newy) coordinate.
    d. It will reset its food-eaten count back to zero (so it must now eat three food objects before it can divide again).
    e. Then skip to step 5.
4. The E. coli will check to see if it overlaps with one or more food objects. If so, it will:
    a. Increase its food eaten count by 1.
    b. Select one of the food objects it overlaps with[14] and tell it to set its state to dead so the food disappears. If it overlaps with more than one food objects, then it must pick one (we don't care which one; don't ask us how to decide) to eat.
5. The E. coli must check to see if it's less than or equal to 256 pixels away from Socrates. If so:
    a. It will get the direction/angle *theta* toward Socrates.
    b. Repeat the following up to ten times:
        i. Try to move forward 2 pixels in the direction of *theta* toward Socrates. If the movement is successful, then return immediately.
        ii. Otherwise, if the E. coli is blocked by dirt, then increase *theta* angle x by ten degrees (wrapping around at 360 degrees).
    c. If the E. coli cannot find a move after ten tries, it stays still.

<u>What E. coli Must Do In Other Circumstances</u>

- E. coli can be damaged by spray and flames. If a spray or flame hits an E. coli, then the E. coli must:
    - Reduce its hit points by the specified amount (as dictated by the spray/flame).
    - If the E. coli is still alive then it must play a sound of SOUND_ECOLI_HURT.
    - Otherwise if a spray or flame reduces the E. coli's hit points to 0 or below (killing it), the E. coli must:

[14] Same definition of overlap as used with flames, spray, etc.

    - Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    - Play a sound effect to indicate that the E. coli died: SOUND_ECOLI_DIE.
    - Increase the player's score by 100 points by using the *StudentWorld::increaseScore()* method.
    - There is a 50% chance the E. coli will turn into food before it dies. If so, the E. coli will introduce a new food object in its current (x,y) coordinate by adding it to the *StudentWorld* object.
- E. coli do not block other objects from moving near or onto them.

Object Oriented Programming Tips

Before designing your base and derived classes for Project 3 (or for that matter, any other school or work project), make sure to consider the following best practices. These tips will help you not only write a better object oriented program, but also help you get a better grade on P3!

Try your best to leverage the following best practices in your program, but don't be overly obsessive – it's rarely possible to make a set of perfect classes. That's often a waste of time. Remember, the best is the enemy of the good (enough).

Here we go!

1. **You MUST NEVER use the imageID (e.g., IID_SALMONELLA, IID_PLAYER, IID_DIRT, etc.) to determine the type of an object or store the imageID inside any of your objects as a member variable. Doing so will result in a score of ZERO for this project.**

2. **Avoid using dynamic cast to identify common types of objects. Instead add methods to check for various <u>classes of behaviors</u>:**

Don't do this:

```
void decideWhetherToAddOil(Actor* p)
{
    if (dynamic_cast<BadRobot*>(p) != nullptr ||
        dynamic_cast<GoodRobot*>(p) != nullptr ||
        dynamic_cast<ReallyBadRobot*>(p) != nullptr ||
        dynamic_cast<StinkyRobot*>(p) != nullptr)
        p->addOil();
}
```

Do this instead:

```
void decideWhetherToAddOil (Actor* p)
{
      // define a common method, have all Robots return true, all
      // biological organisms return false
   if (p->requiresOilToOperate())
        p->addOil();
}
```

3. Always avoid defining specific isParticularClass() methods for each type of object. Instead add methods to check for various common behaviors that span multiple classes:

Don't do this:

```
void decideWhetherToAddOil (Actor* p)
{
       if (p->isGoodRobot() || p->isBadRobot() || p->isStinkyRobot())
              p->addOil();
}
```

Do this instead:

```
void decideWhetherToAddOil (Actor* p)
{
    // define a common method, have all Robots return true, all
    // biological organisms return false
    if (p->requiresOilToOperate())
      p->addOil();
}
```

4. If two related subclasses (e.g., SmellyRobot and GoofyRobot) each directly define a member variable that serves the same purpose in both classes (e.g., m_amountOfOil), then move that member variable to the common base class and add accessor and mutator methods for it to the base class. So the Robot base class should have the m_amountOfOil member variable defined once, with getOil() and addOil()functions, rather than defining this variable directly in both SmellyRobot and GoofyRobot.

Don't do this:

```
class SmellyRobot: public Robot
{
       …
  private:
    int m_oilLeft;
};

class GoofyRobot: public Robot
{
       …
  private:
    int m_oilLeft;
};
```

Do this instead:

```
class Robot
{
  public:
    void addOil(int oil) { m_oilLeft += oil; }
    int getOil() const { return m_oilLeft; }
  private:
    int m_oilLeft;
};
```

5. **Never make any class's data members public or protected. You may make class constants public, protected or private.**

6. **Never make a method public if it is only used directly by other methods within the same class that holds it. Make it private or protected instead.**

7. **Your StudentWorld methods should never return a vector, list or iterator to StudentWorld's private game objects or pointers to those objects. Only StudentWorld should know about all of its game objects and where they are. Instead StudentWorld should do all of the processing itself if an action needs to be taken on one or more game objects that it tracks.**

Don't do this:

```
class StudentWorld
{
  public:
    vector<Actor*> getActorsThatCanBeZapped(int x, int y)
    {
       …      // create a vector with a actor pointers and return it
    }
};

class NastyRobot
{
  public:
    virtual void doSomething()
    {
             …
      vector<Actor*> v;
      vector<Actor*>::iterator p;

      v = studentWorldPtr->getActorsThatCanBeZapped(getX(), getY());
      for (p = actors.begin(); p != actors.end(); p++)
        p->zap();

       }
};
```

Do this instead:

```
class StudentWorld
{
  public:
    void zapAllZappableActors(int x, int y)
    {
```

```
      for (p = actors.begin(); p != actors.end(); p++)
        if (p->isAt(x,y) && p->isZappable())
          p->zap();
    }
};

class NastyRobot
{
  public:
    virtual void doSomething()
    {
              …
      studentWorldPtr->zapAllZappableActors(getX(), getY());
    }
};
```

8. If two subclasses have a method that shares some common functionality, but also has some differing functionality, use an auxiliary method to factor out the differences:

Don't do this:

```
class StinkyRobot: public Robot
{
       …
  public:
    virtual void doDifferentiatedStuff()
    {
      doCommonThingA();
      passStinkyGas();
      pickNose();
      doCommonThingB();
    }
};

class ShinyRobot: public Robot
{
       …
  public:
    virtual void doDifferentiatedStuff()
    {
      doCommonThingA();
      polishMyChrome();
      wipeMyDisplayPanel();
      doCommonThingB();
    }
};
```

Do this instead:

```
class Robot
{
  public:
    virtual void doSomething()
    {
        // first do the common thing that all robots do
      doCommonThingA();

        // then call a virtual function to do the differentiated stuff
      doDifferentiatedStuff();
```

```
        // then do the common final thing that all robots do
      doCommonThingB();
    }

  private:
    virtual void doDifferentiatedStuff() = 0;
};

class StinkyRobot: public Robot
{
      …
  private:
       // define StinkyRobot's version of the differentiated function
    virtual void doDifferentiatedStuff()
    {
        // only Stinky robots do these things
      passStinkyGas();
      pickNose();
    }
};

class ShinyRobot: public Robot
{
      …
  private:
      // define ShinyRobot's version of the differentiated function
    virtual void doDifferentiatedStuff()
    {
        // only Shiny robots do these things
      polishMyChrome();
      wipeMyDisplayPanel();
    }
};
```

Yes, it is legal for a derived class to override a virtual function that was declared private in the base class. (It's not trying to *use* the private member function; it's just defining a new function.)

Don't know how or where to start? Read this!

When working on your first large object oriented program, you're likely to feel overwhelmed and have no idea where to start; in fact, it's likely that many students won't be able to finish their entire program. Therefore, it's important to attack your program piece by piece rather than trying to program everything at once.

Students who try to program everything at once rather than program incrementally almost always fail to solve CS32's project 3, so don't do it!

Instead, try to get one thing working at a time. here are some hints:

1. When you define a new class, try to figure out what public member functions it should have. Then write dummy "stub" code for each of the functions that you'll fix later:

```
class Foo
{
  public:
       int chooseACourseOfAction() { return 0; }      // dummy version
};
```

Try to get your project compiling with these dummy functions first, then you can worry about filling in the real code later.

2. Once you've got your program compiling with dummy functions, then start by replacing one dummy function at a time. Update the function, rebuild your program, test your new function, and once you've got it working, proceed to the next function.
3. **Make backups of your working code frequently. Any time you get a new feature working, make a backup of all your .cpp and .h files just in case you screw something up later.**

BACK UP YOUR .CPP AND .H FILES TO A REMOVABLE DEVICE OR TO ONLINE STORAGE EVERY TIME YOU MAKE A MEANINGFUL CHANGE!

WE WILL NOT ACCEPT EXCUSES THAT YOUR HARD DRIVE/COMPUTER CRASHED OR THAT YOUR CODE USED TO WORK UNTIL YOU MADE THAT ONE CHANGE (AND DON'T KNOW WHAT CAUSED IT TO BREAK).

If you use this approach, you'll always have something working that you can test and improve upon. If you write everything at once, you'll end up with hundreds of errors and just get frustrated! So don't do it.

Building the Game

The game assets (i.e., image and sound files) are in a folder named *Assets*. The way we've written the main routine, your program will look for this folder in a standard place (described below for Windows and macOS). A few students may find that their environment is set up in a way that prevents the program from finding the folder. If that happens to you, change the string literal `"Assets"` in *main.cpp* to the full path name of wherever you choose to put the folder (e.g., `"Z:/CS32Project3/Assets"` or `"/Users/fred/CS32Project3/Assets"`).

To build the game, follow these steps:

For Windows

Unzip Kontagion-skeleton-windows.zip archive into a folder on your hard drive. Double-click on Kontagion.sln to start Visual Studio.

If you build and run your program from within Visual Studio, the Assets folder should be in the same folder as your *.cpp* and *.h* files. On the other hand, if you launch the program by double-clicking on the executable file, the Assets folder should be in the same folder as the executable.

For macOS

Unzip Kontagionskeleton-mac.zip archive into a folder on your hard drive.
Double-click on our provided Kontagion.xcodeproj to start Xcode.

If you build and run your program from within Xcode, the Assets directory should be in the directory yourProjectDir*/DerivedData/*yourProjectName*/Build/Products/Debug* (e.g., */Users/fred/Kontagion/DerivedData/Kontagion/Build/Products/Debug*). On the other hand, if you launch the program by double-clicking on the executable file, the Assets directory should be in your home directory (e.g., */Users/fred*).

What to Turn In

Part #1 (20%)

Ok, so we know you're scared to death about this project and don't know where to start. So, we're going to incentivize you to work incrementally rather than try to do everything all at once. For the first part of Project 3, your job is to build a really simple version of the Kontagion game that implements maybe 15% of the overall project. You must program:

1. A class that can serve as the base class for all of your game's actors (e.g., Socrates, all types of bacteria, goodies, projectiles, etc.):
    i. It must have a constructor that initializes the object appropriately.
    ii. It must be derived from our *GraphObject* class.
    iii. It must have a member function named *doSomething()* that can be called to cause the actor to do something.
    iv. You may add other public/private member functions and private data members to this base class, as you see fit.
2. A dirt class, derived in some way from the base class described in 1 above:
    i. It must implement the specifications described in the dirt pile section above.
    ii. You may add any public/private member functions and private data members to your dirt class as you see fit, so long as you use good object oriented programming style (e.g., you **must NOT** duplicate functionality across classes).

3. A limited version of your Socrates class, derived in some way from the base class described in 1 above (either directly derived from the base class, or derived from some other class that is somehow derived from the base class):
    i. It must have a constructor that initializes Socrates – see Socrates section for more details on where to initialize Socrates.
    ii. It must have an Image ID of IID_PLAYER.
    iii. It must have a limited version of a *doSomething()* method that lets the user pick a direction by hitting a directional key. If the player hits a directional key during the current tick, your code must update Socrates's location appropriately, moving him clockwise or counterclockwise around the Petri dish. All this *doSomething()* method has to do is properly adjust Socrates's (x, y) coordinates using the *GraphObject* class's *moveTo()* method, and our graphics system will automatically animate its movement it around the Petri dish!
    iv. You may add other public/private member functions and private data members to your Socrates class as you see fit, so long as you use good object oriented programming style (e.g., you must not duplicate functionality across classes). But you need not implement firing, spraying, etc. for this part of the project.
4. A limited version of the *StudentWorld* class.
    i. Add any private data members to this class required to keep track of all game objects (right now all of those game objects will just be dirt, but eventually it'll also include bacteria, pits, projectiles like spray/flames, goodies, etc.) as well as your Socrates object. You may ignore all other items in the game such as salmonella, projectiles, goodies, etc. for Part #1.
    ii. Implement a constructor for this class that initializes your data members.
    iii. Implement a destructor for this class that frees any remaining dynamically allocated data, if any, that has not yet been freed at the time the *StudentWorld* object is about to destroyed.
    iv. Implement the *init()* method in this class. It must create Socrates and insert his into the Petri dish at the proper starting location. It must also create all of the dirt piles and add them to the Petri dish as specified in the spec. Your *init()* method may ignore any other objects like food, pits, etc. - it must only deal with Socrates and dirt piles.
    v. Implement the *move()* method in your *StudentWorld* class. During each tick, it must ask Socrates and other actors (just dirt piles for now) to do something. Your *move()* method need not check to see if Socrates has died or not; you may assume for Part #1 that Socrates cannot die. Your *move()* method does not have to deal with any actors other than Socrates and the dirt piles.
    vi. Implement a *cleanup()* method that frees any dynamically allocated data that was allocated during calls to the *init()* method or the *move()* method (i.e., it should delete all your allocated dirt piles and Socrates). Note: Your *StudentWorld* class must have both a destructor and the

cleanUp() method even though they likely do the same thing (in which case the destructor could just call *cleanup()*).

As you implement these classes, repeatedly build your program – you'll probably start out with lots of errors… Relax and try to remove them and get your program to run. (Historical note: A UCLA student taking CS 131 once got 1,800 compilation errors when compiling a 900-line class project written in the Ada programming language. His name was Carey Nachenberg. Somehow he survived and has lived a happy life since then.)

You'll know you're done with Part #1 when your program builds and does the following: When it runs and the user hits Enter to begin playing, it displays a Petri dish with Socrates in its proper starting position and a bunch of dirt piles randomly distributed within the Petri dish. If your classes work properly, you should be able to move Socrates around the Petri dish using the directional keys.

Your Part #1 solution may actually do more than what is specified above; for example, if you are making good progress, try to add food objects to your program. Just make sure that what you have builds and has at least as much functionality as what's described above, and you may turn that in instead.

Note, the Part #1 specification above doesn't require you to implement any E. coli, salmonella, goodies, projectiles like spray or flames, pits, fungi, etc. (unless you want to). You may do these unmentioned items if you like but they're not required for Part #1. **However, if you add additional functionality, make sure that your Socrates, dirt piles, and *StudentWorld* classes still work properly and that your program still builds and meets the requirements stated above for Part #1!**

If you can get this simple version working, you'll have done a bunch of the hard design work. You'll probably still have to change your classes a lot to implement the full project, but you'll have done most of the hard thinking.

What to Turn In For Part #1

You must turn in your source code for the simple version of your game, which **must build without errors** under either Visual Studio or Xcode. We may also devise a simple test framework that runs under g32; if we do, your code **must build without errors** in that framework. If it does not also run without errors, that indicates some fundamental problem that will probably cost you a lot of points. You will turn in a zip file containing nothing more than these four files:

```
Actor.h             // contains base, Socrates, and dirt pile class declarations
                    //     as well as constants required by these classes
Actor.cpp           // contains the implementation of these classes
StudentWorld.h      // contains your StudentWorld class declaration
StudentWorld.cpp    // contains your StudentWorld class implementation
```

You will not be turning in any other files – we'll test your code with our versions of the the other .cpp and .h files. **Therefore, your solution must NOT modify any of our files or you will receive zero credit!** (Exception: You may modify the string literal `"Assets"` in *main.cpp*.) You will not turn in a report for Part #1; we will not be evaluating Part #1 for program comments, documentation, or test cases; all that matters for Part #1 is correct behavior for the specified subset of the requirements.

Part #2 (80%)

After you have turned in your work for Part #1 of Project 3, we will discuss one possible design for this assignment. For the rest of this project, you are welcome to continue to improve the design that you came up with for Part #1, **or you can use the design we provide**.

In Part #2, your goal is to implement a fully working version of Kontagion game, which adheres exactly to the functional specification provided in this document.

What to Turn In For Part #2

You must turn in your source code for your game, which **must build without errors** under either Visual Studio or Xcode. We may also devise a simple test framework that runs under g32; if we do, your code **must build without errors** in that framework. If it does not also run without errors, that indicates some fundamental problem that will probably cost you a lot of points. You will turn in a zip file containing nothing more than these five files:

```
Actor.h             // contains declarations of your actor classes
                    //     as well as constants required by these classes
Actor.cpp           // contains the implementation of these classes
StudentWorld.h      // contains your StudentWorld class declaration
StudentWorld.cpp    // contains your StudentWorld class implementation

report.docx, report.doc, or report.txt  // your report (10% of your grade)
```

You will not be turning in any other files – we'll test your code with our versions of the the other .cpp and .h files. **Therefore, your solution must NOT modify any of our files or you will receive zero credit!** (Exception: You may modify the string literal `"Assets"` in *main.cpp*.)

You must turn in a report that contains the following:

1. A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host

class; also explain why (or why not) you decided to make each function virtual or pure virtual. For example, "I chose to define a pure virtual version of the sneeze() function in my base Actor class because all actors in Kontagion are able to sneeze, and each type of actor sneezes in a different way."

2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. "I didn't implement the Flame class." or "My aggressive salmonella doesn't work correctly yet so I treat it like a regular salmonella right now."
3. A list of other design decisions and assumptions you made; e.g., "It was not specified what to do in situation X, so this is what I decided to do."
4. A description of how you tested each of your classes (1-2 paragraphs per class).

FAQ

Q: The specification is silent about what to do in a certain situation. What should I do?
A: Play with our sample program and do what it does. Use our program as a reference. If neither the specification nor our program makes it clear what to do, do whatever seems reasonable and document it in your report. **If the specification is unclear, but your program behaves like our demonstration program, YOU WILL NOT LOSE POINTS!**

Q: What should I do if I can't finish the project?!
A: Do as much as you can, and whatever you do, make sure your code builds! If we can sort of play your game, but it's not complete or perfect, that's better than it not even building!

Q: Where can I go for help?
A: Try TBP/HKN/UPE – they provide free tutoring and can help you with your project!

Q: Can I work with my classmates on this?
A: You can discuss general ideas about the project, but don't share source code with your classmates. Also don't help them write their source code.

GOOD LUCK!