Louise Lehman
Smallberg
Project 3 (Part 2)

1
```
class MyWorld : public World
{
  public:
      virtual GameStatus RunLevel();
```

This function follows the pseudocode given in the spec. Utilizing a pointer to the NachMan object, first it sets NachMan's state to false, which translates to mean he's not dead, then sets his x and y coordinates to the starting x and y coordinates and changes his direction to NONE. Then it initializes each of the four monsters, one by one, using a for loop and a pointer to each monster. In the loop, each monster's state is set to NORMAL and its x and y coordinates are set to the starting x and y coordinates given in Maze.h. Then it displays the game. To play the game, a while loop calls the DoSomething() functions of all the actors which makes them move and behave appropriately. The last statement in the loop displays the game again, with the players at their new coordinates, in their new states, with pellets erased, if any were eaten. The loop repeats until NachMan is dead or there are no more pellets to eat. Then the RunLevel function returns which was the case to terminate the game play loop, FINISHED_LEVEL or PLAYER_DIED.

RunLevel is pure virtual because its only implementation is written by the student. It is in the derived class MyWorld instead of the parent class World so students can write, change, and/or add code that would be used by Maze.h without making any changes in the other areas of the Maze.h file.

```
class MyMaze: public Maze
{
 public:
      MyMaze(GraphManager* gm)
            : Maze(gm)
      {}
       ~MyMaze();
```

All the MyMaze constructor does is pass a pointer to the GraphManager to the Maze constructor. It was given as such, and I made no changes to it because none were needed. The destructor does nothing.

```
       virtual bool LoadMaze(const std::string &sMazeFile);
```

The LoadMaze function first calls Maze's LoadMaze function so that it constructs the game's maze from the maze text file. This is so that the functions in Maze.h can be used to determine the grid contents and starting locations of the actors by simply calling functions from Maze.h that return these values. Then determineDistances(), a private member function, is called, which assigns to each square in the grid the number of steps, or squares, that square is from the Monsters' starting coordinates. Then it returns true if NachMan's coordinates are not x = -1 and y = -1, which is always true.

LoadMaze is virtual so students can add and implement functions that are accessed by the Maze file without changing anything else in the Maze file. It can then call the function that creates the array holding the distances of each grid square from the Monsters' starting coordinates, which is only accessible in the MyMaze files, and so must be created with a function that is also in MyMaze. It is not pure virtual because Maze has its own LoadMaze function which translates the text file of the maze into grid contents that are meaningful in the game (i.e. WALL, PELLETS, etc.).

```
       virtual bool GetNextCoordinate(int nCurX, int nCurY, int &nNextX, int &nNextY);
```

GetNextCoordinate is given the x and y coordinates of the location of a Monster and returns false if the Monster is already at its starting coordinates and true if it is not. It is also passed references to the next x and y coordinates to move the Monster to in order to be one step closer to its starting coordinates. It checks one step in each direction (up, down, left, and right of the Monster's current location) if the distance at that location in the grid is further or closer to the starting location. It finds the distance by accessing the distances double array which contains the distances of each pair of coordinates from the starting location. The values of the next coordinates the Monster should move to are the coordinates of the first adjacent sqare in the grid that is found that is closer to the starting position.

GetNextCoordinate had to be virtual or pure virtual. so it could be put in the MyMaze class, a derived class of Maze. This is so the student can only make changes in the MyMaze file and not the Maze file. GetNextCoordinate is pure virtual because its only implementation is provided by the student; Maze does not have nor need its own implementation.

```
class Actor
{
public:
      Actor(int y, int x, colors color, int id, World* wptr);
```

The Actor constructor gives values to the private data members for the Actor's ID and starting x and y coordinates, color, and direction. It also saves a pointer to World in a data member. These are all common attributes of NachMan and Monsters, so they all are private data members in the parent class Actor. It is not virtual because constructors cannot be virtual.

```
      virtual ~Actor();
```

The Actor destructor does nothing. It is virtual because destructors of parent classes are always virtual.

```
      int GetX() const;
```

GetX returns the x coordinate, the horizontal component, of the Actor. It is a member function of the Actor class because all Actors have an x coordinate stored in the same data member. It is not virtual because it returns the same private data member for all Actors (NachMan and Monsters).

```
      int GetY() const;
```

GetY returns the y coordinate, the vertical component, of the Actor. It is a member function of the Actor class because all Actors have a y coordinate stored in the same data member. It is not virtual because it returns the same private data member for all Actors (NachMan and Monsters).

```
      void SetX(int newXCoord);
```

SetX changes the x coordinate of the Actor. It is a member function of the Actor class because all Actors have their x coordinate stored in the same data member, so when their x coordinate is to be changed, the same data member is changed for all Actors. It is not virtual because all Actors set a new x coordinate in the same way, by assigning it a new value.

```
      void SetY(int newYCoord);
```

SetY changes the y coordinate of the Actor. It is a member function of the Actor class because all Actors have their y coordinate stored in the same data member, so when their y coordinate is to be changed, the same data member is changed for all Actors. It is not virtual because all Actors set a new y coordinate in the same way, by assigning it a new value.

```
	virtual void DoSomething() = 0;
```

DoSomething basically can make the Actors move, and by doing so they may also change state, color, or direction. NachMan can also change his score and the number of lives he has, and Monsters their target coordinates, vulnerable ticks, and last direction. DoSomthing changes the state of the game by making changes to the Actors and the grid contents.
DoSomething is in the Actor class because all Actors do something during each tick, even though what each Actor does is different.
DoSomething is pure virtual because no Actor moves using the same algorithm. Each Actor will implement their own DoSomething function, and there is not enough commonality between what NachMan will do in its DoSomething function and what Monsters will do in their DoSomething functions to implement a DoSomething function for Actors.

```
	colors GetDisplayColor() const;
```

GetDisplayColor returns the current color of the Actor to be displayed in the game. This function is not virtual and is part of the Actor class because all Actors store their current color in the same varialbe, which is a private data member of Actor, and so return the same variable when their GetDisplayColor function is called.

```
	void ChangeColor(colors c);
```

ChangeColor assigns a new color to the Actor. This function is not virtual and is part of the Actor class because all Actors store their color in the same variable and so change the same variable when changing their color. Therefore, they all change their color the same way.

```
	int getMyID() const;
```

getMyID returns the ID of the Actor. It is a member function of the Actor class because all Actors have an ID stored in the same data member. It is not virtual because it returns the same private data member for all Actors.

```
	Direction direc() const;
```

direc returns the current direction the Actor is moving in, NORTH, SOUTH, EAST, WEST, or NONE. It is a member function of the Actor class because all Actors have the same possible directions they can be moving in and all store their direction in the same private data member in the Actor class. It is not virtual because the same variable is returned for each Actor.

```
	void changedir(Direction d);
```

changedir assigns a new direction to the Actor. This function is not virtual and is part of the Actor class because all Actors store their direction in the same variable and so change the same variable when changing their direction. Therefore, they all change their direction the same way.

```
	World* Worldptr() const;
```

Worldptr returns the pointer to the World class. All Actors point to the same World and store their pointer to World in the same private data member in the Actor class, which was stored when the Actor was constructed. So the same data member is returned for each Actor, and the function doesn't need to be virtual.

```
class NachMan : public Actor
{
public:
	NachMan(World* wptr, int y, int x);
```

The NachMan constructor passes values for the starting x and y coordinates, color, ID, and pointer to World to the Actor class where they will be stored and accessed from. It also sets its score to zero, lives to 3, and state to false, meaning not dead. These last three private data members of the NachMan class are not in the Actor class instead because Monsters don't have scores, lives, and the states of Monsters are not the same as those of NachMan. It is not virtual because constructors cannot be virtual and it wouldn't need to be anyway since it has no derived classes.

```
	~NachMan();
```

The NachMan destructor does nothing. It is virtual so that when Actors are deleted, NachMan's destructor is called along with the Actor's destructor. It is also virtual because NachMan's destructor might be different from a Monster's destructor.

```
	int GetNumLivesLeft() const;
```

GetNumLivesLeft returns the current number of lives NachMan has remaining. This is part of the NachMan class because only NachMan has a life count and so only NachMan would need to return his number of lives. It does not need to be virtual because NachMan has no derived classes.

```
	void DecrementNumLives();
```

DecrementNumLives lessens NachMan's life count by 1 and checks if his number of lives has reached zero. If so, it sets NachMan's state to true, which means dead. It is part of the NachMan class because only NachMan has a life count and so only NachMan would need to decrement his number of lives. It does not need to be virtual because NachMan has no derived classes.

```
	int GetScore() const;
```

GetScore returns NachMan's score, which is stored in one of his private data members. It is part of the NachMan class because only NachMan has a score count and so only NachMan would need to return a score. It does not need to be virtual because NachMan has no derived classes.

```
	void ChangeScore(int s);
```

ChangeScore adds the number of points s to the current score value. It is part of the NachMan class because only NachMan has a score count and so only NachMan would need to add to a score. It does not need to be virtual because NachMan has no derived classes and also is not virtual because it is not overriding a ChangeScore function in the Actor class.

```
	virtual void DoSomething();
```

NachMan's DoSomething function moves in the direction inputted by the player using the arrow keys (NORTH=up, SOUTH=down, EAST=right, WEST=left) or continues in the direction most recently inputted by the player. It only moves NachMan in this indicated direction if there is no wall or cagedoor one square over in that direction. If there is one of these obstructions in the way, NachMan's direction is set back to his previous direction, and NachMan moves in this old direction if there is no obstruction. If there is an obstruction, NachMan does not move, but sits waiting for a valid direction to go in which would be inputted by the player using the arrow keys.
NachMan then checks if the square he is at has any pellets, and if it does NachMan eats it and is awarded

points, the pellet is delted from the grid, and the correct sound is played depending on the type of pellet eaten. If it was a POWERPELLET, the Monsters' states are set to vulnerable if they are not currently in the RETURNTOHOME or MONSTERDIE states. The Monsters' vulnerability count is then set or reset (if they were already in the VULNERABLE state) to the correct value depending on the level being played according to this equation: vulnerability count = 100-level*10 for level 8 and below, and 20 ticks for level 9 and above. DoSomething is derived from the Actor class, in which it is a pure virtual function, because all Actors have a DoSomething function that is implemented in a different way.

```
	bool NMstate() const;
```

NMstate returns true if NachMan is dead, and false if he is alive. This boolean value is stored in a private data member of the NachMan class. Because only NachMan has a state of dead or alive, this value is stored in a private data member in the NachMan class and so the function NMstate, which returns this value, is part of the NachMan class. The function does not need to be virtual because NachMan has no derived classes, and it is not virtual becaus it was not inherited from Actor, since NachMan's state is specific to NachMan.

```
	void changeNMstate(bool st);
```

changeNMstate changes NachMan's state to the state passed to the function, st. Since NachMan's state is specific to NachMan, (Monsters' states are not defined by boolean values and are not dead or alive,) this function is only in the NachMan class. It is not virtual because NachMan has no derived classes and this function wasn't inherited from the Actor class.

```
class Monster : public Actor
{
public:
	Monster(World* wptr, int x, int y, int m_id, colors Mcolor);
```

The Monster constructor stores the starting x and y coordinates of the Monster, its starting (NORMAL state) color, ID, and a pointer to World in private data members. It also assigns zero to the vulnerability tick count and target x and y coordinates, assigns NONE as the last direction, Mcolor to the normal color private data member, and assigns its state to NORMAL. All this information is stored in private data members. It is not virtual because constructors cannot be virtual.

```
	virtual ~Monster();
```

Monster's destructor does nothing. It is virtual because the destructors of base classes are always virtual and when an individual Monster is deleted, its destructor and the Monster destructor are called.

```
	void DoSomething();
```

DoSomething determines the state of the Monster (NORMAL, VULNERABLE, MONSTERDIE, or RETURNTOHOME) and depending on its state, calls a function (respectively, normal(), vulnerable(), monsterdie(), or returntohome()) that will make it perform correctly while in that state. If the Monster has its own version of one of these functions, that Monster's version will be called. Otherwise, the Monster class's version will be called. DoSomething is in the Monster class because all Monsters have a DoSomething function that is called during every tick. It is not virtual because all Monsters perform a switch statement on their state to determine which function to call to act or move appropriately. These functions that are called within the DoSomething implementation will cause them to behave differently from one another, but within the DoSomething function they do the same thing.

```
        void MonsterMove();
```

MonsterMove provides the movement algorithm for all Monsters to move either one step closer to their target square, randomly, or in the opposite direction of their last move. All the information about a Monster's coordinates and target coordinates are in private data members in the Monster class, so the function doesn't need to be passed anything. It only moves the Monster if it is in a NORMAL or VULNERABLE states, because otherwise the Monsters should not move during the tick (in the case of being in the MONSTERDIE state) or should move using a different function (in the RETURNTOHOME state). So it checks for this.
It first tries to move the Monster WEST or EAST *if the Monster's target is in a different column than the Monster.* The Monster moves one step in either direction if there is no wall there, the new coordinate would be within the grid's dimensions, and the move would not cause the Monster to change direction (i.e. move WEST after having moved EAST in the Monster's last movement). If these conditions are satisfied, its x coordinate is set to be one more or one less, and the direction it is moving in is stored in a private data member in the Monster class so that it remembers the last direction it moved in the next time the MonsterMove function is called.
If the Monster's x coordinate has not changed (it did not move WEST or EAST), then the Monster tries to move one step NORTH or SOUTH *if its target is not in the same row as it*. It again checks that there is no wall in the space it wants to move, that the new coordinates of the Monster are within the grid, and that it would not be reversing its direction. If these conditinos are satisfied, its y coordinate is set to be one more or one less and the direction its moving in is stored in the private data member in the Monster class.
If the Monster still hasn't moved (its x and y coordinates are the same as before), a random direction is picked for the Monster to move in. If the Monster can't move in this random direction without running into a wall, off the grid, or reversing its direction, then it tries to move in a different direction. It tries to move NORTH, SOUTH, EAST, or WEST until it is able to, at which point it makes that one movement and stops checking the other directions.
If the Monster still hasn't moved (its x and y coordinates haven't changed yet), the Monster must not be able to move without reversing its direction, and so it moves one step in the opposite direction that it moved in last.
MonsterMove is in the Monster class because all Monsters use this movement algorithm when they move in the NORMAL and VULNERABLE states. It is not virtual because no Monster has a MonsterMove function that makes them move differently when they are in the NORMAL and VULNERABLE states than the way other Monsters move in these states. They do move differently when in the RETURNTOHOME state, but their movement is made by calling a different function.

```
        virtual void normal() = 0;
```

normal makes the Monster establish a target and move in a way specific to each Monster when it is in the NORMAL state. It is a pure virtual function because every Monster has a normal function in which each Monster does something different. It is in the Monster class because every Monster must have its own normal function.

```
        virtual void vulnerable();
```

vulnerable has the Monster establish a target and move when it is in the VULNERABLE state. It will assign random coordinates as the Monster's target only if the Monster is not Clyde, because Clyde does not have a random target when it is in the VULNERABLE state. It then moves the Monster using the MonsterMove function and checks if it is on the same square as NachMan. If it is on the same square as NachMan, it changes the Monster's state from VULNERABLE to MONSTERDIE. Then it decrements the vulnerable ticks counter by 1. If the counter has reached zero, the Monster's state changes to NORMAL and its color changes to the color it has in its NORMAL state only if the Monster is not on the same square as NachMan, in which case the Monster's state has already been changed to MONSTERDIE and will be left as such. This is because it is still during the last tick that the Monster should behave as if it is in the VULNERABLE state, and so if it is on NachMan in the last tick, it should die. It is in the Monster class because all Monsters' DoSomething functions call a function called vulnerable when they are in the VULNERABLE state to make the behave appropriately. It is virtual because the derived class Clyde of Monster has its own vulnerable that calls Monster's vulnerable function and does additional stuff, but all other Monsters just use the Monster class's version of this function.

```
	void monsterdie();
```

monsterdie plays the PAC_SOUND_BIG_EAT noise, since a Monster has been eaten and put in the MONSTERDIE state. Then it changes the Monster's color to LIGHTGRAY and its state to RETURNTOHOME. It i in the Monster class because all Monsters do the same thing when they are in the MONSTERDIE state. It is not virtual because no Monster will do anything differently or in addition to what is in Monster's monsterdie function when it is in the MONSTERDIE state.

```
	void returntohome();
```

returntohome sets the Monster's x and y coordinates to a square that moves it one step closer to its home square until it has reached this square, in which case, the next time the returntohome function is called for that Monster, the Monster will already be at the home square. It finds the next coordinates the Monster should be to by calling the GetNextCoordinate function, which returns the correct coordinates. In the case that the Monster is at the home square when returntohome is called, returntohome changes the Monster's state to NORMAL and color to the color it is when it was first initialized in the NORMAL state and calls the normal function. This goes to the normal function (which is different for each Monster) of the Monster that called the returntohome function. Then the Monster moves and behaves as it should in its NORMAL state. After going throught the normal function, it returns to the returntohome function and does nothing else. This function is in the Monster class because all Monsters have a returntohome function that makes them do something in the RETURNTOHOME state. It is not virtual because all Monsters do the same thing when in the RETURNTOHOME state.

```
	void randomtarget();
```

randomtarget randomly generates an x and y coordinate within the grid's dimensions and sets the target x and y coordinates to these random coordinates. It a function in the Monster class because all Monsters may need to generate random target coordinates at some point and all do it the same way. It is not virtual because no Monster randomly generates target coordinates in a different way.

```
	int targetX() const;
```

targetX returns the x coordinate of the Monster's target square. Although each Monster may have a different x coordinate of their targets, they all store the coordinate in the same private data member in the Monster class and so will return the same data member. This function is not virtual because it is not inherited from the Actor class, (since NachMan doesn't have target coordinates, it doesn't need to be in the Actor class,) and all Monsters return the target's x coordinate the same way.

```
	int targetY() const;
```

targetY returns the y coordinate of the Monster's target square. Although each Monster may have a different y coordinate of their targets, they all store the coordinate in the same private data member in the Monster class and so will return the same data member. This function is not virtual because it is not inherited from the Actor class, (since NachMan doesn't have target coordinates, it doesn't need to be in the Actor class,) and all Monsters return the target's y coordinate the same way.

```
	void setTx(int x);
```

setTx changes the x coordinate of the target square of the Monster to the passed value x. This function is in the Monster class because all Monsters store their target's x coordinate in the same private data member in the Monster class and so will change that same private data member the same way. It is not virtual because it is neither inherited from the Actor class nor implemented in a different way for the derived classes of Monster.

```
	void setTy(int y);
```

setTy changes the y coordinate of the target square of the Monster to the passed value y. This function is in the Monster class because all Monsters store their target's y coordinate in the same private data member in the Monster class and so will change that same private data member the same way. It is not virtual because it is neither inherited from the Actor class nor implemented in a different way for the derived classes of Monster.

```
	MonState Mstate();
```

Mstate returns the current state of the Monster. This is part of the Monster class because all Monsters store their current state in the same private data member in the Monster class, and all Monsters have the same possible states (VULNERABLE, NORMAL, MONSTERDIE, or RETURNTOHOME). Therefore, it is not virtual because it is not derived from the Actor class, (because NachMan doesn't have these states and so it would not need to be in the Actor class for the NachMan class to derive it from), and because all Monsters return their state in the same way (by returning the same private data member).

```
	virtual void changeMstate(MonState m);
```

changeMstate sets the Monster's state to the passed state, m, then changes the Monster's color to whatever it should be in that new state. This is LIGHTBLUE for the VULNERABLE state and LIGHTGRAY for the RETURNTOHOME state. It does not change the Monster's color if the new state is MONSTERDIE because that color change happens in the monsterdie function. To change the Monster's color to the correct color in the NORMAL state for that certain monster, it calls the NormalColor function. It is virtual because the Inky class has its own version of changeMstate which also sets Inky's decideticks data member to zero when the new state is NORMAL. Only Inky does this, and all the other Monsters use the Monster class's version of changeMstate. Since other Monsters use the same implementation of changing their state, it is a fuction in the Monster class, and not each individual Monster's class.

```
	void setvticks(int n);
```

setvticks changes the number of vulnerable ticks for the Monster to the passed value n. This changes depending on the level of the game currently being played. All Monsters have a vulnerability count, so the private data member vticks is in the Monster class and this function that alters its value is also in the Monster class. It is not virtual because the Monsters all change vticks the same way.

```
	colors NormalColor();
```

NormalColor returns the color that the Monster calling the function is when it's in the NORMAL state. This color is stored in a private data member in the Monster class during initialization. All Monsters have a color that they are in the NORMAL state, so this function is in the Monster class. It is not virtual because all Monsters return the same private data member when returning their normal color.

```
class Inky : public Monster
{
public:
	Inky(World* wptr, int x, int y);
```

Inky's constructor passes his starting x and y coordinates, color, ID, and a pointer to World to the Monster constructor. It also assigns zero as the value of the counter for every ten ticks he decides whether or not to chase NachMan when he's in the NORMAL state, and assigns the value true to the boolean value chase. This fuction is not virtual because constructors cannot be virutual and Inky passes the Monster constructor different variables for his color and ID than other Monsters do.

```
	~Inky();
```

Inky's destructor does nothing. It is virtual so that when Inky is deleted, both Inky's and the Monster class's destructors are called.

```
	virtual void normal();
```

Inky's normal function sets Inky's target coordinates when Inky is in a NORMAL state, then calls the MonsterMove function in the Monster class to make Inky move towards his target. Every ten ticks, Inky randomly decides whether or not to chase NachMan or pick a random target, with an 80 percent chance of wanting to chase NachMan. So if his counter for these ten ticks reaches zero, it is set to 10 and he randomly makes a decision by generating a random number from 0 to 100. If the number is from 0 to 79, he sets his target coordinates to NachMan's coordinates; if it is 80 to 99, he picks random target coordinates. He then calls the MonsterMove function to make his move. After returning from MonsterMove, he checks if he is on the same square as NachMan (their x and y coordinates are the same). If so, he sets NachMan's state to dead. Lastly, he decrements his deciding ticks counter by 1. This function is derived from the pure virtual funtion in the Monster class. It is in the Monster class because all Monsters have a normal function whose implementation is different from all other Monsters' normal functions. This normal funciton in Inky's class defines what Inky does in the NORMAL state.

```
	virtual void changeMstate(MonState m);
```

changeMstate is virtual because it is inherited from the Monster class. It first calls the Monster class's changeMstate function because Inky will change state just as other Monsters do. But this function also sets a counter to zero for how many ticks have gone by since he's decided whether or not to chase NachMan while he is in the NORMAL state. changeMstate is virtual because only Inky has a counter for this and so only Inky sets this counter to zero when he is put into a NORMAL state. All other Monsters use the same version of this function, so it is in the Monster class (but Inky has his own version in his class).

```
class Stinky : public Monster
{
public:
	Stinky(World* wptr, int x, int y);
```

Stinky's constructor passes his starting x and y coordinates, color, ID, and a pointer to World to the Monster constructor. This fuction is not virtual because constructors cannot be virutual. Stinky also has a starting color and ID different from the other Monsters.

```
	~Stinky();
```

Stinky's destructor does nothing. It is virtual so that when Stinky is deleted, both Stinky's and the Monster class's destructors are called.

```
	virtual void normal();
```

Stinky's normal function sets Stinky's target coodinates to NachMan's coordinates if NachMan is within five columns or less and five rows or less of Stinky. If NachMan is not within this range of Stinky, then Stinky sets his target coordinates to random coordinates by calling the randomtarget function. Then Stinky's normal function calls the MonsterMove function to move Stinky. After returning from MonsterMove, Stinky checks if he has the same coordinates as NachMan. If so, he sets NachMan's state to dead (true). This function is derived from the pure virtual funtion in the Monster class. It is in the Monster class because all Monsters have a normal function whose implementation is different from all other Monsters' normal functions. This normal function in Stinky's class defines what Stinky does in the NORMAL state.

```
class Dinky : public Monster
{
public:
	Dinky(World* wptr, int x, int y);
```

Dinky's constructor passes his starting x and y coordinates, color, ID, and a pointer to World to the Monster constructor. This fuction is not virtual because constructors cannot be virutual. Dinky also has a starting color and ID different from the other Monsters.

```
	~Dinky();
```

Dinky's destructor does nothing. It is virtual so that when Dinky is deleted, both Dinky's and the Monster class's destructors are called.

```
	virtual void normal();
```

Dinky's normal function sets Dinky's target coordinates, moves Dinky by calling the MonsterMove function, then checks if Dinky has the same x and y coodinates as NachMan.
Dinky checks if his x coordinate is the same as NachMan's x coordinate. If so, they are in the same column. If there are no walls between Dinky and NachMan in this column, Dinky sets his target's x and y coordinates to NachMan's coordinates. A loop checks if there is a wall at each pair of coordinates between Dinky and NachMan in this column. It does this by initializing a variable i to zero then setting it to Dinky's y coordinate plus 1 if Dinky's column is less than NachMan's and to Dinky's y coordinate minus 1 if his y coordinate is greater than NachMan's. Every iteration of the loop, i gets closer to NachMan's y coordinate by 1 and the grid contents are checked for a wall at the value of i for y and Dinky and NachMan's shared x coordinate for x. If i equals NachMan's y coordinate or a wall is found, the loop ends. Then i is checked for equality to NachMan's y coordinate, and if they are equal, Dinky's target coordinates are set to NachMan's coordinates.
If Dinky's x coordinate is not the same as NachMan's x coordinate, their y coordinates are checked for equality. If they are equal, Dinky and Nachman are in the same row. If there are no walls between Dinky and NachMan in this row, Dinky sets his target's x and y coordinates to NachMan's coordinates. A loop like the one before checks for walls, except instead of i representing the y coordinate, it represents the x coordinate. If i equals NachMan's x coordinate at the end of the loop, Dinky's target coordinates are set to NachMan's coordinates.
If Dinky and NachMan are not in either the same column or the same row, or they are but there is at least one wall between them, then Dinky calls the randomtarget function to set give his target square random coordinates.
Then Dinky moves, and if Dinky has the same coordinates as NachMan after moving, he sets NachMan's state to dead (true).
This function is derived from the pure virtual funtion in the Monster class. It is in the Monster class because all Monsters have a normal function whose implementation is different from all other Monsters' normal functions. This normal function in Dinky's class defines what Dinky does in the NORMAL state.

```
class Clyde : public Monster
{
public:
      Clyde(World* wptr, int x, int y);
```

Clyde's constructor passes his starting x and y coordinates, color, ID, and a pointer to World to the Monster constructor. This fuction is not virtual because constructors cannot be virutual. Clyde also has a starting color and ID different from the other Monsters.

```
      ~Clyde();
```

Clyde's destructor does nothing. It is virtual so that when Clyde is deleted, both Clyde's and the Monster class's destructors are called.

```
      virtual void normal();
```

Clyde's normal function sets Clyde's target coordinates randomly by calling the randomtarget function. Then Clyde moves by calling the MonsterMove function. Lastly, Clyde checks if he has the same coordinates as NachMan, and if so, sets NachMan's state to dead (true). This function is derived from the pure virtual funtion in the Monster class. It is in the Monster class because all Monsters have a normal function whose implementation is different from all other Monsters' normal functions. This normal function in Clyde's class defines what Clyde does in the NORMAL state.

```
      virtual void vulnerable();
```

Clyde's vunerable function determines which quadrant of the grid NachMan is in, and sets his target coordinates to the coordinates of the furthest corner from NachMan in the quadrant diagonally across from NachMan. For example, if NachMan's x coordinate is less than or equal to the maze width divided by two, and NachMan's y coordinate is less than or equal to the maze height divided by two, then NachMan is in the upper left quadrant of the grid. In this case, Clyde's target coordinates would be the lower right corner of the grid - the target x coordinate is one less than the maze width, and the target y coordinate is one less than the maze height. If NachMan is in the upper right quadrant of the grid, Clyde's target is the lower left corner of the grid, etc. Then the vulnerable function of the Monster class is called. This allows Clyde to do what all the other Monsters do when they're vulnerable, except for the fact that Clyde's target coordinates are not random, whereas those of all other Monsters when in the VULNERABLE state are. (As a reminder: in the Monster class's vulnerable function, Clyde will move then check if he is on the same square as NachMan and decrement the counter for the number of ticks he is vulnerable.)
This function is virtual because three of the Monsters use the same implementation (the implementation in the Monster class) for what to do when they are in the VULNERABLE state, but Clyde does something a little differently when he is in teh VULNERABLE state. Therefore, only Clyde needs to override Monster's vulnerable function and all other Monsters will inherit the implementation of the vulnerable function in the Monster class. This is also why it must be defined in the Monster class and not only the individual monsters' classes; so its implementation for the three other Monsters need not be written multiple times.

2
As far as I know, everything is implemented and works correctly. I am not able to view NachMan's score or the level because the text goes off the screen and I cannot extend the screen far enough to compensate for it. So those values might not be correct, but I am assuming they are.

3
I was not sure what should happen in the Monster::vulnerable() function when the vulnerability ticks reaches zero if NachMan and the Monster are on the same square. Either the Monster is still vulnerable and

dies and awards points to NachMan then checks if his vulnerability ticks has reached zero. Or the ticks are checked first and the Monster is changed to a NORMAL state, in which case NachMan dies and the Monster is still in a NORMAL state. So what I did was check if the Monster is on the same square as NachMan first, and if so award points and put the Monster in the MONSTERDIE state; then decrement the ticks and if the tick count is zero, only change the Monster's state to NORMAL if it is not on NachMan so the Monster hasn't just died.
I also assume nothing needs to be done in the destructors since no memory is dynamically allocated. So my destructors do nothing.

4
For the Actor class

For the NachMan class

For the Monster class
I printed out the vulnerability ticks to see if all four Monsters had a count and decremented it once per tick.

For the MyWorld class

For the MyMaze class