1. Public Function Explanation

----Actor----

Actor(int id, int sx, int sy, colors sColor, World\* ptr);

-The constructor actor. Made for the sole purpose of initializing all the private data members that are common between all of Actor’s subclasses. All actors have an id, coordinates, a color, and a pointer to the World class.

virtual ~Actor();

-The virtual destructor. Polymorphism is used so it is best if there is a virtual destructor. This is the same for all other class’ destructors.

int GetX() const;

int GetY() const;

colors GetDisplayColor() const;

int getMyID() const;

World\* getWorld() const;

-Simple getter functions within the Actor class that return the values of the various private data members. These must be in Actor class since the Actor is the only one that can access these private members. None of these are virtual since they are never overwritten later in the class hierarchy.

virtual void DoSomething() = 0;

-The generic movement function for all subclasses for Actor. Since all Actors have the ability to move, this is places in the Actor class. Also this is made pure virtual since we don’t want generic Actors being constructed and since generic Actors should not be able to do anything. This also forces DoSomething to be redefined for every subclass for Actor.

virtual void completeReset() = 0;

-All Actors must completely reset after the player has died or a new maze is loaded, thus is placed within the Actor base class. Also, this is made pure virtual since an Actor can’t do a completeReset but its two subclasses are forced to.

void SetX(int newXCoord);

void SetY(int newYCoord);

void setColor(colors newColor);

-Simple setter functions to change the various private data members for the Actor class. No need to overwrite these in later classes thus not virtual.

-----NachMan-----

NachMan(World\* wPtr, int sx, int sy);

-Constructor as specified by the specs.

~NachMan();

-Non-virtual since no other classes are derived from NachMan.

int GetNumLivesLeft() const;

int GetScore() const;

bool isAlive() const;

-Simple getter functions again.

void DecrementNumLives();

void setDead();

void setAlive();

void setDirection(Direction dir);

void incrementScore(int aScore);

-Simple setter functions to change NachMan’s own private member variables.

virtual void DoSomething();

-NachMan’s DoSomething function that overrides Actor’s pure virtual function. NachMan’s DoSomething function first checks if NachMan’s current location contain any pellets or power pellets. It increments the NachMan’s score accordingly and changes all Monsters to a vulnerable state(setMonsterToVulnerable). Then it checks if NachMan is on the same tile as any or all of the monsters and acts accordingly (uses helper function –NachManMeetsMonster- to check this). If NachMan sets itself dead do to being eaten, then it returns so that it cuts NachMan’s DoSomething function short. NachMan then checks if there is food remaining in the maze and if there is, then it takes in a direction from the player. If the player can move in that direction(canMoveInNextDirection) then it sets NachMan’s coordinates one step in that direction(setNextCoord). If it can’t move in that direction, it checks if NachMan’s previous direction is available to be moved in and moves one tile in that direction. If it can’t move in either direction, then it sets NachMan’s direction to NONE and does not change NachMan’s coordinates at all.

virtual void completeReset();

-Resets NachMan’s starting position, resets him back to alive, resets his direction to NONE.

-----Monster-----

Monster(int id, int sx, int sy, colors sColor, World\* ptr, monsterStatus status);

-Monster’s constructor takes in the first 5 parameters in order to use Actor’s constructor. It takes in a monsterStatus in order to initialize Monster’s m\_status private member.

virtual ~Monster();

-Virtual destructor due to Monster having subclasses.

virtual void DoSomething();

-All monsters also have a DoSomething function. DoSomething is defined for all Monsters since they have the same general algorithm for DoSomething. However, it is still virtual since one of the monsters(Inky) must redefine DoSomething in order to work properly. This general DoSomething first checks if NachMan is in the Monster’s tile. If it is, depending on the Monster’s status, it either eats NachMan and returns, thus cutting off the DoSomething function early to prevent extra movement, or Monster has itself eaten and changes NachMan’s stats accordingly(all using MonsterMeetsNachMan). If it doesn’t detect NachMan on the same tile, then it moves according to its current status. If NORMAL status, DoSomething uses the monster’s moveNormal function and sets the Monster’s pastX and pastY values using setPastX/Y. If VULNERABLE status, use moveVulnerable, if MONSTERDIE, use moveMonsterDie, if RETURNTOHOME, then use moveReturnToHome.

void nextCoordToTarget(int tx, int ty, int& fx, int& fy);

-This is the basic movement algorithm that takes in a target coordinate and “returns” the coordinate of the next step towards the target coordinate using fx and fy reference integers. This function uses the algorithm given within the specs. In order to check all four Directions randomly to choose from (if it could not properly move according to the target’s direction) all four Directions are put into a vector. An integer is then set to 4 and modulus with this integer is used to determine a number from 0 to that integer minus 1. Then it uses this determined number to get a Direction from the vector and attempts to move one step in that Direction. It returns(to “break” the function) if able to move in randomly determined Direction. If unable to move in that Direction, then it erases that Direction from the vector and decrements the integer used in the modulus operation. While the vector isn’t empty, it repeats the Direction-choosing process above. If it manages to exit the while loop, then the monster can only move back to the tile it came from and sets its coordinates accordingly.

All Monsters and only Monsters use this class, thus it is defined in the Monster class. Also, it is not redefined and overwritten in any other class thus not virtual.

void moveRandom();

-This function generates a random x and y coordinate using rand and modulus to generate a random coordinate. If the random coordinate is the same as the Monster’s current coordinates, it generates another set. It then uses nextCoordToTarget in order to set the Monster’s x and y coordinates. All Monsters and only Monsters use this class, thus it is defined in the Monster class. Also, it is not redefined and overwritten in any other class thus not virtual.

virtual void moveNormal() = 0;

-All Monsters can move while in the NORMAL state and thus this function is defined in the Monster class. However, since all Monsters have completely different movement algorithms during NORMAL status, it is pure virtual so that all monsters are forced to define their own moveNormal function. The moveNormal functions in each Monster are only there purely as movement algorithms due to the complexity of these algorithms. Unlike the other moveXXX functions, the moveNormal function will not generally change the state of the Monster except for its coordinates. Inky must change one of its other states while moving in the NORMAL state but due to this limitation I set upon this function, I had to virtual the DoSomething function to take care of this.

virtual void moveVulnerable(colors clr);

-All Monsters can move while in the VULNERABLE state and thus this function is defined in the Monster class. While VULNERABLE, a monster generally chooses a random target to move to(using moveRandom) and decrements its vulnerable ticks count. If that tick count reaches 0, then this function sets the Monster back to NORMAL state, resets the tick count, resets the Monster’s color(thus the colors parameter), and calls moveNormal so that the Monster starts moving normally again. This function is virtual since Inky must tweak this function a little to work properly and Clyde uses a completely different algorithm.

void moveMonsterDie();

-All Monsters behave in exactly the same way when in MONSTERDIE status thus it is defined in monster and is not virtual. This function simply sets the Monster into RETURNTOHOME status, plays the proper sound, and sets the Monster to the proper color.

virtual void moveReturnToHome(colors clr);

-All Monsters have a RETURNTOHOME algorithm thus this function is defined in the Monster base class. Generally, this function uses MyMaze’s GetNextCoordinate function to check if the Monster is back at its starting coordinates. If it is, then this function reverts the Monster back to NORMAL status and color(thus the parameter), and resets its vulnerability ticks. It then calls the Monster’s moveNormal function. If the Monster isn’t at the home tile, then it simply sets the Monster’s x and y coordinate to the coordinates that GetNextCoordinate “returns”.

void MonsterMeetNachMan(NachMan\* nmPtr);

-All Monster use this function to check if they have run into NachMan and thus it is defined in Monster and is non-virtual. It checks if the Monster’s current coordinates match NachMan’s. If they do, then the Monster either eats NachMan and returns(if its current state is NORMAL), or it lets itself get eaten by NachMan and decrements NachMan’s score accordingly and changes its state to MONSTERDIE(if its current state is VULNERABLE).

virtual void completeReset();

-All Monsters need to be reset if NachMan has died or if new maze is loaded thus placed in this function. Virtual because Inky needs to add few more lines of code to it. This general version resets the Monster’s status back to NORMAL, resets its vulnerability ticks back to its max value, and resets its color back to its original value.

void setStatus(monsterStatus status);

void decrementTicks();

void resetTicks();

void setPastX(int px);

void setPastY(int py);

-Simple setter functions

monsterStatus GetStatus() const;

int getTicks() const;

-Simple getter functions.

-----Inky-----

Inky(World\* ptr, int sx, int sy);

-Constructor as specified by the specs.

~Inky();

-Non-virtual destructor since there are no subclasses for Inky

virtual void DoSomething();

-Virtual for good practice. Essentially the same as the general DoSomething function from the Monster class except it must also handle the decrementing and resetting of Inky’s decision ticks while in NORMAL state. While in NORMAL state, Inky checks if it has to decide again. If he does, he resets his decision ticks, generates his decision and calls moveNormal, which moves him accordingly. He then remembers his previous coordinates. If he doesn’t, he simple uses moveNormal and decrements his decision ticks. While in all the other states, he calls upon the appropriate moveXXX function and remembers his previous coordinate if necessary.

virtual void moveNormal();

-Virtual for good practice. Inky checks if he wants to chase NachMan. If he does, he gets NachMan’s coordinates using the World pointer and uses them in MyMaze’s GetNextCoordinate function. He then sets his coordinates to the coordinates the function returns. If he doesn’t, he simply calls upon moveRandom.

virtual void moveVulnerable(colors clr);

-Virtual for good practice. It is the same algorithm as the moveVulnerable function defined in the Monster base class except it must also reset Inky’s decision ticks if it’s vulnerability ticks reach 0. Unfortunately, since this must be done before moveNormal is called, moveVulnerable must be redefined since the ticks can’t be reset after moveVulnerable is called(this would give Inky an extra tick before his next decision).

virtual void moveReturnToHome(colors clr);

-Virtual for good practice. Just like moveVulnerable, it uses the same algorithm as the one defined in Monster except it must also reset Inky’s decision ticks once Inky reaches his home tile. Once again, the ticks must be reset before moveNormal is called, forcing me to redefine moveReturnToHome.

virtual void completeReset();

-Virtual for good practice. Calls Monster’s completeReset function and also resets Inky’s decision and decision ticks.

----Stinky----

Stinky(World\* ptr, int sx, int sy);

-Constructor as specified in the specs.

~Stinky();

-Non-virtual destructor due to lack of subclasses.

virtual void moveNormal();

-Virtual for good practice. This function first establishes four variables dictating how far Stinky can sniff in the four directions. In all four directions, while he hasn’t reached the end of the grid or his five-tile sniffing limit, Stinky moves the four variables one more coordinate away from himself. At the end of this checking, Stinky now knows the limits of how far he can sniff. He then investigates each coordinate in between the top-leftmost limit to the bottom-rightmost limit. If he finds NachMan, he sets the two target variables to NachMan’s coordinates. If those target variables turn out to be 0 after all of that, Stinky was not able to sniff NachMan and simple calls moveRandom. If those target variables are any other value, Stinky calls GetNextCoordinate and sets his coordinates to what that function “returns”.

-----Dinky-----

Dinky(World\* ptr, int sx, int sy);

-Constructor designated by specs

~Dinky();

-Non-virtual destructor due to no subclasses for Dinky

virtual void moveNormal();

-Virtual for good practice. Dinky sets up four variables equal to his coordinates(to use to iterate through the four coordinates) as well as two target variables equal to 0. Then it checks all four directions for NachMan. While he hasn’t hit a wall or the edge of the Maze, Dinky advances straight away from his position. If he finds NachMan, he sets the target variables to NachMan’s coordinates, calls GetNextCoordinate and sets his coordinates to what that function outputs. He then returns to “break” the function to prevent unnecessary checking. If he manages to get out of those checks, he has not found NachMan and simply calls moveRandom.

-----Clyde-----

Clyde(World\* ptr, int sx, int sy);

-Constructor as designated by the specs.

~Clyde();

-Non-virtual destructor due to no subclass after Clyde.

virtual void moveNormal();

-Virtual for good practice. Simply calls moveRandom. However, in order to work with the general DoSomething function defined within Monster, this had to be done.

virtual void moveVulnerable(colors clr);

-Virtual for good practice. Clyde first checks if his vulnerability ticks have reached 0. If so, he reverts back to NORMAL state and calls moveNormal and resets his vulnerability ticks. If the ticks haven’t reached 0, then Clyde finds NachMan’s coordinates. He then checks if NachMan is in the left or right of the maze and inside each of these checks, he checks if NachMan is in the upper or lower half. When he finds NachMan, he sets his target coordinate to the opposite corner from NachMan. He then calls GetNextCoordinate and uses what it “returns” in order to set his new coordinates. He decrements his vulnerability ticks and then returns in order to “break” the function and stop unnecessary checking.

----MyWorld----

virtual GameStatus RunLevel();

-Virtual for good practice. RunLevel follows the algorithm provided to us within the specs. I get pointers to NachMan, the Maze, and all the Monster for easier access. I ask all Actor’s to do a completeReset and display the screen. While NachMan isn’t dead and there are still pellets on the screen, then all four actors are asked to doSomething. The screen is displayed again after every iteration of the while loop. Once it checks the while loop, it checks if it is either due to NachMan dying or NachMan eating all pellets. If NachMan is dead, then it returns Player\_Died. If all pellets are eaten, then it loads the next maze.

-----MyMaze-----

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

-Virtual for good practice. Checks if nCurX and nCurY are at the home tile and returns false if so. If not, then it checks all four directions for a tile whose value is exactly one less than it within the distance array. If it finds one, it “returns” that next tile and returns true.

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

-Virtual for good practice. Calls Maze’s LoadMaze function in order to initialize the GridContents array. If Maze’s LoadMaze returns false, then this function will not run determineDistance(algorithm from homework 2 to determine optimum paths). If it returns true, then a GridContents array is initialized and determineDistance is allowed to run.

2. Bugs

-All Actors work as specified by the specs. There seems to be a little bit of lag but this also seems to be present within the .exe file provided to us.

3. Design Decisions

-The biggest design decision I made was for the moveNormal function for all the Monsters. In all other moveXXX functions, I allowed the Monster to change any private data members necessary. However, since most Monster’s behavior during NORMAL status tended to be much more complicated, I decided to have moveNormal be restricted to only changing the Monster’s coordinates and nothing else (if it needed to). This was one reason why Inky had to redefine a separate doSomething function for itself since it had to change one of its other private members besides its coordinates while moving in NORMAL status.

4. Testing

Actor-

-Created simply as a base class. Functions simply set or returned values.

NachMan

-NachMan was vigorously tested in various mazes to see if he responded to key inputs correctly, if he knew where the walls were, to see if he stopped moving if he could not move, and to see if he continued to move in the previous direction if the player inputted an invalid direction. He was also tested to see if he could remove any kind of pellet from any direction he came from. NachMan was then tested with each Monster individually, checking to see if he could recognize them in their various statuses. When Monsters were in a NORMAL state, NachMan was tested to see if he would die no matter what angle or direction the Monster came from. When Monsters were in a VULNERABLE state, NachMan was tested to see if he could eat them from any angle or direction. In both MONSTERDIE and RETURNTOHOME states, NachMan was tested to make sure he never died or ate the monster due to contact. After testing NachMan with each Monster for each status, I then tested NachMan to see if he would properly turn Monsters VULNERABLE if he ate a Power Pellet. Finally, he was tested to once again see if he could eat these Monsters that turned from NORMAL to VULNERABLE status.

Monster classes

-All Monsters were tested individually for each of its four statuses. Each Monster and NachMan were put into specific situations in order to test out their movement algorithms when in NORMAL behavior. They were checked to see if they tracked NachMan only when they were able to, see if they did not reverse their path, and were able to eat NachMan from any angle and direction. They were also checked to see if they really did have random movement while VULNERABLE (or for Clyde’s case, to see if he moved towards the correct corner and hovered around there). They were also checked to see if they were properly eaten by NachMan when VULNERABLE and if they were properly reverted to MONSTERDIE status. Monsters were checked to make sure they did not move the tick they reverted to MONSTERDIE status. Monsters were placed in various parts of the Maze while in RETURNTOHOME status to make sure they took the optimum path back home. I also made sure that all statuses and counts were reset to NORMAL for the Monsters once they reach the home tile.

MyMaze

-The functions in MyMaze were tested by extracting the definition of determineDistances and LoadMaze and putting them into a separate main() inside of another project. They were tweaked so that they worked and I input different mazes imitating the mazes within the txt files. I then checked the distance array that was outputted from determineDistances to make sure the paths were properly calculated. And then Monsters were put in various spots in the maze to make sure they follow the proper path back home to test out the GetNextCoordinate function.

MyWorld

-Tested by killing NachMan and finishing multiple mazes multiple times to make sure that all Actors were properly reset.