Report

1.Every Public member function followed by description

**Actor.h**

Actor Class: public GraphObject

Actor(StudentWorld\* sw, int imageID, int startX, int startY, Direction startDirection, float size, unsigned int depth);

Every Actor was initialized this way because it needed:

1. A parameter that defined which StudentWorld to belong in
2. An image ID to define the Actor’s image
3. A starting and X and Y coordinate as well as a starting Direction
4. A size and depth parameter to set the size of each Actor and which Actor overlapped with the others, basically the interaction of Actors wtith each other.

virtual void getAnnoyed(unsigned pts) =0 ;

This is virtual because all actors would get annoyed. It is an ABC because an actual Actor object will not do anything with getting annoyed. The pts parameter is for how many hitpoints to take off.

virtual void doSomething() = 0;

this is virtual because all actors will do Something at each tick. It is an ABC because the actual actor does not actually doSomething.

void setStatus(bool b);

a set status function for every actor will set the state of the actor to Boolean b so that every tick it will know whether to be deleted or kept alive.

bool isAlive() const;

a check on whether something is alive or not for every actor. It’s const since you don’t change anything

virtual unsigned returnIdentity() = 0;

returnIdentity gives each actor an identity as a number. The identities follow the ID numbers given to us. It is virtual because every actor has different id. It is an ABC because we will never need the ID of an Actor Object itself.

StudentWorld\* getSW() const;

this function gets the studentWorld pointer from the private data members so that we know which student variable we are pointing to.

bool isVisi(){ return visible; };

This is to check if something is visible.

void setVisi(bool b){ visible = b; };

this is a function to set the visibility of some actor object.

bool canMoveTo(Direction dir, int X, int Y);

checks if the actor object can move to that coordinate when it is facing that direction. This is generally used for protesters but my design is to have the derived class of Regular Protester be from the Actor class.

bool canMoveInDir(Direction dir);

checks if the actor object can move in that direction. This function is for frackman and the protesters.

void moveInDir(Direction dir);

moves the actor object in that direction. This function is for squirt, frackman and the protesters.

virtual bool getleavetheOil(){ return false; };

this function is for the protesters. The virtual is because it wants to return whether the protesters are leaving the oil or not. The return false is just in case it is called on the Actor object. It is not an ABC because that would mean we would have to write an implementation for all the other actors which would seem excessive. Instead we put a harmless return false statement that we will never call.

Frackman Class: public Actor

Frackman(StudentWorld\* sw);

Frackman constructor that has the student world it should be pointing to. It is always going to start at 30,60 so no starting coordinates

virtual void getAnnoyed(unsigned pts);

it is virtual to supplant the virtual getAnnoyed of Actor. It decrements the amount of pts assigned to it from its hitpoints.

unsigned getHP(){ return hitPoints; };

returns how much HP is left in the Frackman

unsigned getunitWater(){ return unitWater; };

returns how much squirts it has left in Frackman

unsigned getGold(){ return gold; };

returns how much gold frackman has on him.

unsigned getSonar(){ return sonarCharge; };

returns how much sonars he has.

bool noBoulderBlock(Direction dir);

checks if there are any boulders one space ahead of him. This is for when he is moving in directions. It doesn’t allow him to move left if hes going to hit a boulder.

void incSonarBy(int i){ sonarCharge += i; };

increment sonar count by i.

void incWaterBy(int i){ unitWater += i; };

increment his squirt count by i.

void incGold(){ gold++; };

increment his gold count.

virtual void doSomething();

his doSomething is virtual to supplant the Actor’s doSomething(). He does something every tick that is different from actor’s.

virtual unsigned returnIdentity() { return 0; };

his return Identity is 0. We don’t ever call if but its to prevent the ABC error.

double distanceToBoulder(double x1, double y1, double x2, double y2);

this checks his distance from any of the boulders.

Dirt Class: public Actor

Dirt(StudentWorld\* sw, int startX, int startY);

Dirt constructor that has the studentWorld it points to and the starting coordinates as its parameters.

virtual unsigned returnIdentity() { return 6; };

It is virtual since it is different for all different actors. It will return 6 if we check for any dirt objects.

virtual void getAnnoyed(unsigned pts){};

it is virtual since Actor’s is virtual. It does nothing.

virtual void doSomething(){};

it is virtual since Actor’s is virtual. It does nothing.

Boulder Class: public Actor

Boulder(StudentWorld\* sw, int startX, int startY);

Boulder constructor that has the studentWorld it points to and the starting coordinates as its parameters.

virtual unsigned returnIdentity() {return 4;};

It is virtual since it is different for all different actors. It will return 4 if we check for any Boulder objects.

virtual void getAnnoyed(unsigned pts){};

it is virtual since Actor’s is virtual. It does nothing.

virtual void doSomething();

it is virtual because Actor’s is virtual. It will fall accordingly when all the dirt underneath is dug up.

Squirt Class: public Actor

Squirt(StudentWorld\* sw, int startX, int startY, Direction dir);

Squirt constructor that has the studentWorld it points to and the starting coordinates as its parameters.

virtual unsigned returnIdentity() { return 3; };

It is virtual since it is different for all different actors. It will return 3 if we check for any Squirt objects.

virtual void getAnnoyed(unsigned pts) {};

it is virtual since Actor’s is virtual. It does nothing.

virtual void doSomething();

it is virtual because Actor’s is virtual. It will check if it hit any protesters and decrement their hitpoints.

Barrel Class: public Actor

Barrel(StudentWorld\* sw, int startX, int startY);

Barrel constructor that has the studentWorld it points to and the starting coordinates as its parameters.

virtual unsigned returnIdentity() { return 5; };

It is virtual since it is different for all different actors. It will return 5 if we check for any Barrel objects.

virtual void getAnnoyed(unsigned pts) {};

it is virtual since Actor’s is virtual. It does nothing.

virtual void doSomething();

it is virtual because Actor’s is virtual. If it is within range of the frackman, it will kill itself and then increment points.

Gold Class: public Actor

Gold(StudentWorld\* sw, int startX, int startY, bool v, bool tempState, bool pickUpableF, unsigned ts);

It is the gold nugget class. It has the studentworld it points to, the starting coordinates, a Boolean for visibility, a Boolean for whether it is in its temporary state, a Boolean for whether it is pickupable by frackman, and an unsigned int for how long it will remain in the temporary state, as its parameters. Visibility is starting visibility. If its not initialized as true for temporary state then its in the permanent state. If its not pickupable by frackman then it is by protesters.

virtual unsigned returnIdentity() { return 7; };

It is virtual since it is different for all different actors. It will return 7 if we check for any Gold objects.

virtual void getAnnoyed(unsigned pts){};

it is virtual since Actor’s is virtual. It does nothing.

virtual void doSomething();

it is virtual because the actor’s is virtual. It will check for distances and increment frackman’s gold count accordingly.

Sonar Class: public Actor

Sonar(StudentWorld\* sw, int startX, int startY, unsigned ts);

Sonar constructor that has the studentWorld it points to, the starting coordinates, and how long it will remain alive before it is dead as an unsigned int, as its parameters.

virtual void getAnnoyed(unsigned pts){};

it is virtual since Actor’s is virtual. It does nothing.

virtual unsigned returnIdentity() { return 8; };

It is virtual since it is different for all different actors. It will return 8 if we check for any Sonar objects.

virtual void doSomething();

it will check and reveal the objects within its vicinity

Water Class: public Actor

Water(StudentWorld\* sw, int startX, int startY, unsigned ts);

Water constructor that has the studentWorld it points to, the starting coordinates, and how long it will remain alive before it is dead as an unsigned int, as its parameters.

virtual void getAnnoyed(unsigned pts){};

it is virtual since Actor’s is virtual. It does nothing.

virtual unsigned returnIdentity() { return 9; };

It is virtual since it is different for all different actors. It will return 9 if we check for any Water objects.

virtual void doSomething();

it is virtual because the actor’s is virtual. If it hits any protesters it will stun them and decrement their hitpoints by 2. It will also kill itself after 4 steps.

RegProtester Class: public Actor

RegProtester(StudentWorld\* sw, int startX, int startY, int id, int leftSquares, int tw, int hp);

Regular protester constructor that has the student world it points to, the starting coordinates, the id since it can be hardcore or regular protester, an int that stores how many squares left it has to walk in a current direction, another int to store how many ticks to wait, and a final int to store as its hitpoints, as parameters.

virtual unsigned returnIdentity() { return 1; };

It is virtual since it is different for all different actors. It will return 1 if we check for any RegularProtester objects.

unsigned getHP(){ return hitPoints; };

just returns the HP.

void setHP(int i){ hitPoints = i; };

sets hp to i.

virtual void getAnnoyed(unsigned pts);

if hitpoints less than or equal to 0

leave the oil

ticks to wait = 0

return if pts == 99 //special case when it is annoyed by gold

else

increase score, play sound

else increase score, play sound, and reset ticks to wait

virtual void doSomething();

it is virtual because actor’s is virtual.

Check if its alive

Check if ticks <=0

Else

If leavetheoil == true

goBack to 60,60

else if facing frackman and distance is less than 4

shout and play sound

else if in line of sight, distance greater than 4, and can move there

take step in that direction

else if numsquares < 0

change direction and walk in that direction.

Else if we don’t skip this step then

Check if we can go in a perpendicular direction and do it

Else move in the current direction one step.

void returnArr(char (&arr)[64][64]);

returns an array of chars that will have ‘X’ as points where boulders and dirts are and ‘.’ where they aren’t.

bool explore(queue<Coord>& qu, char(&arr)[64][64], int sr, int sc);

pushes onto the queue if the coordinates sr and sc are not checked yet. The array of chars is to check whether we can or not.

bool checkSquare(int sr, int sc, char (&arr)[64][64]);

just checks a 4 by 4 square with the sr and sc at the bottom left of it. The arr tells us whether there is a dirt or boulder object blocking the way.

bool facingFm();

whether we are facing the frackman

Direction lineOfSight();

Whether the frackman is within our line of sight

int getNumOfLeftSquares(){ return numSquaresToMoveInCurrentDirection; };

returns number of square left to move in that direction.

void setNumOfLeftSquares(int i){ numSquaresToMoveInCurrentDirection = i; };

sets the number of square left to move in that direction.

unsigned getticksToWait(){ return ticksToWait; };

returns ticks to wait before protester acts.

void decticksW(){ ticksToWait--; };

decrements ticks to wait

void setTicksToWait(int i){ ticksToWait = i; };

sets the number of ticks to wait.

void setleavetheOil(bool b){ leavetheOil = b; };

sets whether or not the protester is in the leave the oil state.

virtual bool getleavetheOil(){ return leavetheOil; };

this is virtual because the hardcore protester or the regular protester might not be leaving the oil at the same time of course.

void setperpSec(int i){ perpSec = i; };

sets the number of seconds until we can go perpendicualar

bool canPerp(Direction currDir);

checks whether we can go perpendicular to the current direction.

int getperpSec(){ return perpSec; };

returns the number of seconds until we can go perpendicular direction.

bool goBack();

tells the protester to go back to 60,60

uses map<coordinates, direction> and queues

start at 60,60

while the queue isn’t empty

go in all 4 directions

stores the opposite direction that you went into a map

searches for the current coordinate and then goes for it from the map.

int getshoutSec(){ return shoutSec; };

returns number of seconds until we can shout

void decshoutSec(){ shoutSec--; };

decrements number of seconds until we can shout

void resetshoutSec(){ shoutSec = 15; };

resets number of seconds until we can shout

HardProtester Class: public RegProtester

HardProtester(StudentWorld\* sw, int startX, int startY,int id, int leftSquares, int tw);

Has starting student word, starting coordinates, squares to left to take and ticks to wait, as parameters

The get Annoyed function for the RegProtester works for both.

virtual unsigned returnIdentity(){ return 2; };

it is virtual because actor’s is. Returns 2 when it is called for

virtual void doSomething();

it is virtual because actor’s is. The only difference in doSomething than regular protester is when it has to look for the frackman. That is implemented using a map that stores coordinates and integers. It is used the same way the map with directions is used except you are storing integers as representation of distance now.

**StudentWorld.h**

StudentWorld(string assetDir) : GameWorld(assetDir){};

given

~StudentWorld();

Not used. It will just call the clean up function.

virtual int init();

it will make new barrel, gold and barrel objects. It will initialize the first protester. It will also destroy all the dirt needed to do so.

virtual int move();

just tells all the actors to do something

virtual void cleanUp();

just uses a for loop and destroys al actors

void setDisplayText();

sets the display text.

string setDigits(int item, int length);

just the spacing for digits

string makeDisplayNeat(unsigned score, unsigned level, unsigned lives, unsigned health, unsigned squirts, unsigned gold, unsigned sonar, unsigned barrelsLeft);

a helper function for setDisplayTest.

bool checkforDirt(int lx, int ly);

checks for dirt at that coordinate

bool checkforBoulder(unsigned lx, unsigned ly);

checks for boulder at that coordinate

void destroyDirt(int i, int j);

destroys the dirt at that coordinate

void getActors(unsigned identity, vector <Actor\*> &temp);

uses the identity integer and puts the alive actors into temp

void addActors(int count, char c, int sr = 0, int sc = 0);

adds the actors into the studentworld. C defines which ones to add.

Frackman\* getFm()const{ return fm; };

Returns the frackman of the world

void decBarrels(){numberOfBarrels--;};

decreases the number of barrels by one.

unsigned getnumberOfBarrels(){return numberOfBarrels;};

gives you the number of barrels

double distance(int x1, int y1, int x2, int y2);

calculate the distance between two integers.

2. None.

3. Definitely ambiguous distance from boulder that the frackman could not touch. The spec said 3, but it is definitely not three. I tried to make it as similar to the game as possible so mine is set at 1.5 from the center. The spacing of the display text seemed ambiguous. Never mentioned whether there were spaces after the colons. I just went with however close I could get.

4. I tested my classes every time I implemented them. Most of the objects that would destruct after you got them were easy. For the protesters I used many for loops to see what exactly was going on and how they were moving. Other than that I played the game a lot and saw how they interacted with each other.