1. DESCRIPTION OF PUBLIC MEMBER FUNCTIONS

**// StudentWorld public member functions //**

* StudentWorld(string assetDir) – simple constructor for StudentWorld class
* ~StudentWorld() – StudentWorld’s destructor, deletes all Actors
* virtual void int() – finds the correct level to load, allocates all Actors, sets bonus to 1000,
* virtual void move() – sets the display text at the top, does something for each actor, deletes the actors that are dead, decreases the bonus by 1, exposes the exit if ready, decreases lives if player dies, and declares victory if player finishes the level
* virtual void cleanUp() – similar to destructor, deletes all Actors
* virtual void setDisplayText() – takes the score, level, current bonus, lives, player health, and player ammo and displays text using string streams
* unsigned int decBonus() – if bonus is positive, decreases the bonus by 1
* bool fireUpon(int x, int y) – checks space at (x, y) to see if a shootable Actor is there. if there is, then it shoots the Actor and returns true. else if a wall or factory is there, it will block and return true, else it returns false.
* bool doFactoryCensus(int x, int y, int distance, int& count, Actor\* notMe) – checks each space within the region of [x-distance,x+distance]X[y-distance,y+distance] to count each kleptobot. if there is a robot on the factory, return false. otherwise, the function returns true and sets the reference count to be the number Kleptobots within the region defined above.
* void makeBullet(int x, int y, GraphObject::Direction dir) – allocates a new Bullet, given the current space and direction of the shooter
* void makeGoodie(int x, int y, Kleptobot::GoodieType type) – allocates a new Goodie for when a Kleptobot holding a Goodie is destroyed.
* void makeKleptobot(int x, int y, KleptobotFactory::ProductType type) – allocates a new Kleptobot when factory calls upon it.
* void exposeExitIfReady() – if there are no jewels left and the exit isn’t already revealed, then the exit is revealed and the SOUND\_REVEAL\_EXIT is played.
* bool canPlayerMoveHere(int x, int y) const – goes through each Actor in the StudentWorld’s list to see if the Player can move to the space at (x,y)
* bool canRobotMoveHere(int x, int y) const - goes through each Actor in the StudentWorld’s list to see if a Robot can move to the space at (x,y)
* bool canBoulderMoveHere(int x, int y) const - goes through each Actor in the StudentWorld’s list to see if a Boulder can move to the space at (x,y)
* bool doesSpaceBlockVision(int x, int y) const - goes through each Actor in the StudentWorld’s list to see the space at (x,y) blocks a robot’s vision
* unsigned int getBonus() const – accessor function for bonus
* Actor\* getActorAt(int x, int y, Actor\* notMe) const – returns an Actor at the space (x,y) that is not notMe. If there is no actor at that space, returns a nullptr.
* Player\* getPlayer() const – accessor function for m\_player
* Actor\* getStealable(int x, int y) const – returns an Actor that is stealable at the space (x,y), else returns a nullptr.

**// Actor public member functions //**

* Actor(int imageID, int startX, int startY, Direction dir, StudentWorld\* world) – constructor for Actor which sets its image, starting coordinate (x,y), direction, and a pointer to the StudentWorld that its in. It also makes the GraphObject visible, and sets its m\_alive to true. This constructor will need to be called by an initializer list for its derived classes.
* virtual ~Actor() – simple destructor, just in case
* virtual void doSomething – This is a virtual dummy function. It is virtual because some Actors actually do nothing during the doSomething call such as Walls or Boulders, while other actors have uniquely defined doSomething functions.
* void setDead() – sets the Actor’s m\_alive to false. This function is not virtual because it can be used universally across all of the actors.
* virtual void getShot() = 0 – This is a pure virtual function because each Actor can get shot by a bullet. Each Actor reacts differently when shot, so it makes sense for this function to be virtual.
* virtual bool checkEmptySpace(int x, int y) const – calls upon the canPlayerMoveHere function from the StudentWorld, and returns true if the player can move there.
* bool checkBoulder(int x, int y, Actor\* notMe) const – returns true if the Actor at (x, y) is a Boulder.
* virtual bool blocksPlayer() const – This is a virtual function that returns true by default since many types of Actors do block the Player. It is virtual because there are some classes that do not block Players, such as Goodies, Bullets, and Boulders.
* virtual bool blocksRobot() const – This is a virtual function that returns true by default since many types of Actors do block Robots. It is virtual because there are some classes that do not block Robots, such as Goodies and Bullets.
* virtual bool blocksBoulder() const - This is a virtual function that returns true by default since many types of Actors do block Boulders. It is virtual because there are some classes that do not block Boulders, such as Holes and Bullets.
* virtual bool blocksBullet() const - This is a virtual function that returns false by default since many types of Actors do not block Bullets. It is virtual because there are some classes that do block Bullets, such as Walls and Factories.
* virtual bool blocksVision() const - This is a virtual function that returns false by default since many types of Actors do not block a Robot’s vision. It is virtual because there are some classes that do block Robots’ vision, such as Boulders, Walls, other Robots, and Factories.
* virtual void moveObject(int deltaX, int deltaY, Direction dir) – This is a virtual function that moves an object based on its direction. This function is used Bullets. However, other classes move differently which is why this function is virtual.
* StudentWorld\* getWorld() const – an accessor function that returns a pointer to the Actor’s m\_world.
* bool isAlive() const – checks to see if the Actor is alive
* virtual bool isShootable() const – This is a virtual function that returns true by default since many types of Actors are shootable. It is virtual because there are some classes that cannot be shot such as Goodies, Holes, Walls, and Factories.
* virtual bool isStealable() const – This is a virtual function that returns false by default since most Actors are not stealable by Kleptobots. It is virtual because there are some classes that can be stolen by Kleptobots such as ExtraLives, RestoreHealth, and Ammo Goodies.

**// Player public member functions //**

* Player(int X, int Y, StudentWorld\* world) – constructor for Player, initializes its starting point (x,y), and StudentWorld. The Actor constructor is called in its initializer list, giving it the proper image and the right direction. The player is also initialized with 20 ammo, 20 health, and its victory condition is set to false.
* virtual ~Player() – simple destructor, defined just in case
* virtual void doSomething() – checks if the player is alive, and does the action appropriate to the key pressed. The player is set dead for ESC, the player shoots for SPACE, and moves accordingly (if it can) to the ARROW KEY pressed.
* virtual void moveObject(int deltaX, int deltaY, Direction dir) – The player moves changes its direction and moves to the space in that direction if that space does not have an Actor that blocks the Player.
* virtual void getShot() – decrements the player’s health by 2 points. If the player still has health, the SOUND\_PLAYER\_IMPACT is played. Else, the player is dead and the SOUND\_PLAYER\_DIE is played.
* void shoot() – a function unique to the Player. If the Player has ammo, a bullet is shot, the SOUND\_PLAYER\_FIRE is played, and the Player’s ammo is decremented by 1.
* void loadAmmo() – used when an Ammo Goodie is consumed, adds 20 ammo.
* void restoreHealth() – used when a RestoreHealth Goodie is consumed, restores the player’s health back to 20
* void finishLevel() – called by the Exit’s doSomething, sets the Player’s m\_victory to true.
* int getHealth() const – accessor function that returns the Player’s m\_health
* int getAmmo() const – accessor function that returns the Player’s m\_ammo
* bool getVictory() const – accessor function that returns the Player’s m\_victory

**// Wall public member functions //**

* Wall(int X, int Y, StudentWorld\* world) – constructor for Wall, sets its starting point (x,y), image, and world.
* virtual ~Wall() – simple destructor, just in case
* virtual bool blocksBullet() const – returns true, since Walls block bullets
* virtual bool blocksVision() const – returns true, since Walls block vision
* virtual bool isShootable() const – returns false, since Walls are not shootable
* virtual void getShot() – does nothing, since Walls do nothing when they are shot

**// Boulder public member functions //**

* Boulder(int X, int Y, StudentWorld\* world) – constructor for Boulder, setting its starting point (x,y), image, and world. Its health is also initialized to 10.
* virtual ~Boulder() – simple destructor, defined just in case
* virtual void getShot() – decrements its health, and sets it to dead if it has no health left
* virtual void moveObject(int deltaX, int deltaY, Direction dir) – if there is an empty space where the boulder wants to move, it will move there.
* virtual bool blocksPlayer() const – returns false, since Boulder does not block the Player
* virtual bool blocksVision() const – returns true, since Boulder blocks vision
* virtual bool checkEmptySpace(int x, int y) const – checks if a Boulder can move to the spot (x,y). It is different than the Actor’s function because it must check if the spot has an Actor that blocks the Boulder, not the Player.

**// Bullet public member functions //**

* Bullet(int X, int Y, Direction dir, StudentWorld\* world) – constructor for Bullet, settings its starting point (x,y), image, and world
* virtual ~Bullet() – simple destructor, defined just in case
* virtual bool isShootable() const – returns false, because Bullets are not shootable
* virtual bool blocksPlayer() const – returns false, because Bullets do not block Players
* virtual bool blocksBoulder() const – returns false because Bullets do not block Boulders
* virtual bool blocksRobot() const – returns false because Bullets do not block Robots
* virtual void doSomething() – checks if it can fireUpon anything at its current location, moves in the direction that it was shot in, and checks if it can fireUpon anything at its new location. It will set itself to dead if it hits anything shootable.
* virtual void getShot() – does nothing, since bullets do not get shot

**// Hole public member functions //**

* Hole(int X, int Y, StudentWorld\* world) – constructor for Holes, setting its starting (x,y) coordinate, image, and world
* virtual ~Hole() – simple destructor, defined just in case
* virtual void getShot() – does nothing, since Holes cannot get shot
* virtual void doSomething() – checks if a Boulder is on it. if there is, then the Hole and Boulder are both set to dead (i.e. the Boulder is swallowed by the Hole)
* virtual bool blocksBoulder() const – returns true since Holes do not block Boulders
* virtual bool isShootable() const – returns false since Holes are not shootable

**// Goodie public member functions //**

* Goodie(int image, int X, int Y, StudentWorld\* world, int points) – constructor for the base class, Goodies, setting their (x,y) starting space, image, points, and world
* virtual ~Goodie() – simple destructor, defined just in case.
* virtual void getShot() – does nothing since Goodies do nothing when shot
* virtual bool isShootable() const – returns false since Goodies cannot be shot
* virtual bool blocksPlayer() const – returns true since Goodies do not block Players
* virtual bool blocksRobot() const – returns true since Goodies do not block Robots
* virtual void doSomething() – if the Goodie is alive and the player is standing on it, the Goodie will be set to dead, the appropriate amount of points will be added to the score, the SOUND\_GOT\_GOODIE will play, and the doExtra function will be called (its effect is different for each type of goodie0
* virtual void doExtra() = 0 – this function is pure virtual since all Goodies have a different extra effect for the player.
* void reveal() – sets the Goodie to visible.
* void hide() – sets the Goodie to invisible.
* bool isRevealed() const – accessor function for the Goodie’s m\_visible.

**// Jewel public member functions //**

* Jewel(int X, int Y, StudentWorld\* world) – constructor for Jewels, setting their (x,y), image, points, and world.
* virtual ~Jewel() – does nothing, defined just in case
* virtual void doExtra() – does nothing since Jewels have no extra effect

**// Exit public member functions //**

* Exit(int X, int Y, StudentWorld\* world) – constructor for Exit, setting its (x,y) starting point, image, points, and world, set to invisible by default
* virtual ~Exit() – does nothing, defined just in case
* virtual void doSomething() – if the Player is on the Exit and the Exit is visible, the points are increased, the SOUND\_FINISHED\_LEVEL is played, and the Player’s victory condition is set to true, ending the level.
* virtual void doExtra() – does nothing

**//**  **ExtraLife public member functions //**

* ExtraLife(int x, int y, StudentWorld\* world) – constructor for ExtraLife, setting its (x,y) starting point, image, points, and world
* virtual ~ExtraLife() – does nothing, defined just in case
* virtual void doExtra() – increases the life by one
* virtual bool isStealable() const – returns true since ExtraLives are stealable by Kleptos

**// RestoreHealth public member functions //**

* RestoreHealth(int x, int y, StudentWorld\* world) – constructor for RestoreHealth, setting its (x,y) starting point, image, points, and world
* virtual ~RestoreHealth() – does nothing, defined just in case
* virtual void doExtra() – restores the player’s health back to 20
* virtual bool isStealable() const - returns true since RestoreHealths are stealable by Kleptos

**// Ammo public member functions //**

* Ammo(int x, int y, StudentWorld\* world) – constructor for Ammo, setting its (x,y) starting point, image, points, and world
* virtual ~Ammo() – does nothing, defined just in case
* virtual void doExtra() – gives the player 20 more ammo
* virtual bool isStealable() const – returns true since Ammos are stealable by Kleptos

// **Robot public member functions //**

* Robot(int image, int x, int y, Direction dir, StudentWorld\* world, int health, int points) – constructor for Robots, a base class.
* virtual ~Robot() – deconstructor, does nothing, defined just in case
* virtual void doSomething() = 0 – Pure virtual function since each type of Robot does something differently.
* virtual void getShot() – decreases a Robot’s health by 2 if the Robot still has health, plays SOUND\_ROBOT\_IMPACT and returns. if there is no health left, the Robot is setDead, SOUND\_ROBOT\_DIE is played, and appropriate amount of points is added to the score
* void incTurn() – increases the number of the turn for the robot by 1.
* virtual void moveObject(int deltaX, int deltaY, Direction dir) – defines the movement function for Snarlbots, if the space is empty, they move there. if not, they move in the opposite direction.
* virtual bool checkEmptySpace(int x, int y) const – checks if there is an Actor at (x,y) and if that Actor blocks Robot. if nothing is there or it does not block Robots, returns true.
* bool playerInSight() const – It is not virtual since it is used by all Robots. If the Robot is facing the Player and the Player is in the same row/column, and each space between them does not block the Robot’s vision returns true. Else false.
* virtual bool blocksVision() const – returns true since Robots block each other’s vision
* int getTurn() const – gives the current turn
* int getTicks() const – gives the current tick

**// Snarlbot public member funtions //**

* Snarlbot(int x, int y, Direction dir, StudentWorld\* world) – derived class from Robot, initializes (x,y) starting space, direction, points, and world
* virtual ~Snarlbot() – does nothing, defined just in case
* virtual void doSomething() – if the Snarlbot is allowed to doSomething on this tick, then it will check if the player is in sight and shoot accordingly, else it will move / turn.

**// KleptoBot public member functions //**

* enum GoodieType { NONE, AMMO, RESTORE\_HEALTH, EXTRA\_LIFE} – defines the type of Goodie, a KleptoBot can hold
* Kleptobot(int x, int y, StudentWorld\* world, int imageID, int health, int points) – derived from Robot, initializes (x,y) starting space, points, and world
* virtual ~Kleptobot() – drops the Goodie it is holding
* virtual void doSomething() – if the Klepto can move on this tick, it will either shoot the player, steal a goodie, or move accordingly
* void pickUp(Goodie\* ap) – deletes the Goodie and remembers the type it holds
* void drop() – if it has one, it will allocates a new Goodie in the spot where the Robot dies
* virtual bool isShootingRobot() const = 0 – a pure virtual function since There are two types of Kleptobots

**// RegularKleptoBot public member functions //**

* RegularKleptobot(int x, int y, StudentWorld\* world) – derived from Kleptobot
* virtual ~RegularKleptobot() – does nothing,
* virtual bool isShootingRobot() const – returns false, since it does not shoot

**// AngryKleptoBot public member functions //**

* **AngryKleptoBot** (int x, int y, StudentWorld\* world) – derived from Kleptobot
* virtual ~ **AngryKleptoBot** () – does nothing,
* virtual bool isShootingRobot() const – returns true, since it does shoot

**// KleptoBotFactory public member functions //**

* enum ProductType { REGULAR, ANGRY} – defines if it produces Regular or Angry Kleptobots
* KleptobotFactory(int x, int y, StudentWorld\* world, ProductType type) - constructor
* virtual ~KleptobotFactory() – does nothing, defined just in case
* virtual void doSomething() – produces a Klepto if the census returns true and there are less than 3 in the region
* virtual void getShot() – does nothing
* virtual bool isShootable() const – returns true since it is shootable
* virtual bool blocksBullet() const – returns true since it blocks bullets
* virtual bool blocksVision() const – returns true since it blocks vision

2. I believe that my program functions just as the solution one does, hopefully, without mistakes.

3. DESIGN DECISIONS:

First, I decided to use a list instead of a vector. At first I tried using vectors, but I realized that the address changes and shuffles each time something is added or deleted, so this caused some problems for me. I then switched to linked lists to hold my Actors, and it worked out fine.

Secondly, I decided to make certain classes base classes, such as Robot, Goodie, and KleptoBot. These were very helpful since they all had sub classes that were very similar to each other.

Third, I wasn’t sure exactly how the KleptoBots moved if they were trapped. So I made it so that they change to a random direction, and cannot move.

4. TESTING

I tested the StudentWorld class by messing around with each level, and making my own levels. This was easy enough to test.

I tested my Player class by moving boulders, shooting, and getting shot. Sometimes, I would change the space bar to test other functions than shooting. For example, I would test the Player’s get victory function by pressing the space bar, and I would win. This was very helpful for me, in testing all of the Player’s functions.

I tested my Wall class simply by running into the Wall to make sure the Player could not move into it. I would also shoot the Wall to ensure that it blocked Bullets. Also, I pushed boulders to ensure that it blocked Boulders. I also made sure that Robots could not walk into Walls either. The Walls were pretty easy to test.

I tested my Boulder and Hole classes by making my own level of Boulders and Holes. I pushed the Boulders into things and made sure that if they were blocked by Boulders, then I couldn’t move it there. Similarly I pushed Boulders into Holes several times to make sure the Hole and Boulder both died. I also made sure that Actors (except for Bullets) could not move onto Holes.

I tested my Bullet class by shooting several things. I shot things that were not shootable such as Goodies and Holes and things that were shootable such as Walls and Robots. I made sure that the exact outcome occurred. I also made sure my getShot functions for each class did the correct thing. I even initialized Boulders to Bullets once in one of the premade levels, which was weird, but the outcome did work.

I tested all my Goodie classes by stepping on them and making sure the outcome of each type of Goodie was correct. I made sure that the KleptoBots could only steal the right ones. Additionally, I made sure that they disappeared when I stepped on them. The Exit was particularly hard, since we could not use the isVisible function, but I just made my own member variable for visibility and it worked out fine.

I tested each Robot individually. I made sure they couldn’t move where they shouldn’t such as into Walls or Factories or Holes. The hardest part for me, was probably the playerInSight function. I made sure this worked by testing it several times on my own levels. The Snarlbots were simple, but the KleptoBots (both angry and regular) were a bit of a pain. I took away their random chance to ensure that they could pick up Goodies. And I surrounded them by Boulders to check their movement.

Lastly, I tested the Robot Factories by taking away the random chance and ensuring that they made Robots. This helped me a lot. The factoryCensus function was particulary hard, but I was able to test it by making my own levels.