* bool spotPassable(int destX, int destY, Actor \*p, list<Actor \*> &orig)
  + Not a member function b/c I had issues getting it to work in any class.
  + Pseudocode:
  + Base case: if p is null, return true
  + Otherwise, create a pointer to object at destination by calling getObjectAt
    - If it’s not a nullptr, get the type of that object
    - If the type is an impassable object, return false
    - If the type is a passable object
      * Create a copy of actors without that object
      * Return true & call spotPassable again with the new list & object
  + Return false

**Actor Class**

* int getItemType() const { return m\_itemType; }
  + Not virtual because this is usable by all actors. Simply returns what type the actor is.
* virtual int getHP() const { return m\_hp; }
  + Virtual b/c it was simpler to implement this way with my class hierarchy. I simply initialized the hp of all objects besides damageable actors to 0. It would have been more efficient to make this pure virtual, but I ran out of time to make this change. Simply returns the HP of the actor.
* virtual void decHP() { m\_hp -= 2; }
  + Virtual b/c it was simpler to implement this way with my class hierarchy. Function is only ever called for damageable actors. It would have been more efficient to make this pure virtual, but I ran out of time to make this change. Simply decreases HP by 2.
* virtual void incHP(int x) { m\_hp += x; }
  + Virtual b/c it was simpler to implement this way with my class hierarchy. Function is only ever called for the player. It may have been more efficient to make this pure virtual, but I ran out of time to make this change. Simply increases HP by x amount, and is only called when the Restore HP goodie is collected.
* virtual int getAmmo() const { return m\_ammo; }
  + Virtual b/c it was simpler to implement this way with my class hierarchy. I simply initialized the anno of all objects besides ones who could attack to 0. It would have been more efficient to make this pure virtual, but I ran out of time to make this change. Simply returns the ammo of the actor.
* virtual void incAmmo(int x) { m\_ammo += x; }
  + Virtual b/c it was simpler to implement this way with my class hierarchy. Function is only ever called for the player. It may have been more efficient to make this pure virtual, but I ran out of time to make this change. Simply increases ammo by x amount, and is only called when the ammo goodie is collected.
* virtual void decAmmo() { m\_ammo--; }
  + Virtual b/c it was simpler to implement this way with my class hierarchy. Function is only ever called for actors that can attack. It would have been more efficient to make this pure virtual, but I ran out of time to make this change. Simply decreases HP by 2.
* virtual int characterMove(int destX, int destY) { return -2; }
  + Virtual b/c it must be redefined for the character class (for use by the player) & I ran out of time to fix this. Should be more efficient as pure virtual. Function is only ever called for the player.
* virtual void collectItem(int type, Actor \*item) { return; }
  + Virtual b/c it needs to be redefined for players and there were issues making it pure virtual. Simply returns
* virtual void damage(int type) { return; }
  + Virtual b/c it needs to be redefined for damageable actors. Issues making it pure virtual. Simply returns.
* virtual void doSomething() { return; }
  + Virtual b/c it needs to be redefined for actors who can do something. Otherwise, it simply returns for objects that do nothing during a tick such as walls.
* virtual void moveInDir(Direction dir);
  + Virtual b/c it was simpler to implement this way with my class hierarchy and I ran out of time to change it. Simply calculates the destination if an object moves in a certain direction, then moves in that direction.
* virtual void initMembers(bool state, int imageID);
  + Virtual b/c it was simpler to implement this way w/ my class hierarchy and I ran out of time to check if I could do it another way. Simply sets the visibility and alive state when an object is created, as well as uses the image ID to identify what type of object it is. Function created b/c exits don’t start as visible.

**Exit Class**

* virtual void doSomething();
  + Virtual b/c it needs to be redefined for this class. Simply checks if all jewels have been collected, and opens the exit if it hasn’t been opened yet, then plays correct sound.

**Bullet Class**

* virtual void doSomething();
  + Virtual b/c it needs to be redefined for this class. Simply checks if bullet is alive, then calls the bulletCollision function to see if the bullet collides with anything.
  + Returns

**KBotFactory Class**

* virtual void doSomething();
  + Virtual b/c it needs to be redefined for this class. Simply checks region to see if there are more than 3 bots in it or if there is a bot on the factory. If there isn’t it has a 1/50 chance of spawning either a Kbot or angry kbot.

**Character Class**

* virtual int characterMove(int destX, int destY);
  + Virtual b/c it needs to be accessed through an Actor pointer for the player.
  + Psesudocode:
  + If space is empty at destination, move to that space
  + Else check if that space is passable
    - If the space is passable, move to location and check if there is a collectable item
      * If an item is collectable, collect it
      * Return
    - Otherwise, check if the player destination is a boulder
      * Check if the destination of the boulder is empty and move there if it is
      * Else, check if it’s a hole, then push and fill the hole
      * Return
* virtual void collectItem(int type, Actor \*item);
  + Virtual b/c it needs to be accessed through an Actor ptr and is redefined for characters. Function is only ever called by the player.
  + Pseudocode:
    - If the item is dead/not visible, return
    - Otherwise, identify what type of item it is
      * Depending on item, increase ammo/HP/Life and score
      * Then, set the item to dead, and play item collected sound
* virtual void damage(int type);
  + Virtual b/c it needs to be redefined for damageable characters. Later redefined for Kleptobots b/c it worked better this way w/ my class hierarchy
  + Pseudocode:
  + First, decrease HP
  + Identify what type of object it is (player, boulder, snarlbot)
    - Player: If HP<0, set player to dead and play death sound
      * Otherwise, play player impact sound
    - Boulder: If HP<0, set boulder to dead
    - Snarlbot: If HP<0, set bot to dead, increase score, then play death sound
      * Otherwise, play impact sound
* virtual void fireBullet();
  + Virtual b/c it needs to be redefined for characters.
  + Pseudocode:
    - If no more ammo, simply return
    - Otherwise decrease ammo
    - Identify if shooter is player or robot, and play appropriate sound
    - Identify what direction the shooter is facing, and create a new bullet object on that location
    - Return

**Player Class**

* virtual void doSomething();
  + Virtual b/c it needs to be redefined for players. Pseudocode:
  + If player is dead, return
  + Otherwise, get input if there is any
    - For UDLR, simply call the characterMove function for the dest that direction, then set the direction to that
    - For space, call the fireBullet function
    - For ESC, simply set alive state to false
  + Return

**Robot Class**

* virtual bool canFireAtPlayer();
  + Virtual b/c it needs to be accessed through an actor pointer.
  + Pseudocode:
  + Check if all spots in the direction the robot is facing is passable using spotPassable
    - If a spot isn’t passable, check if the player is there
    - If it is a player, fire a bullet and return
* virtual void initTicks(int x) { ticks = curTicks = x; }
  + Virtual because it needs to be accessed through an actor pointer. Simply initializes the ticks and current ticks value..
* virtual int getTicks() const { return curTicks; }
  + Virtual b/c it needs to be accessed through an actor pointer. Simply returns the current ticks.
* virtual void decTicks() ;
  + Virtual b/c it needs to be accessed through an actor pointer. Simply decreases the current ticks.

**Snarlbot Class**

* virtual void doSomething();
  + Virtual b/c it needs to be redefined for Snarlbots.
  + Pseudocode:
  + Check if ticks have been initialized, and initialize if they have not been
  + Return if the snarlbot is dead
  + Otherwise check if current ticks = 0
    - Check if the Snarlbot can fire at player using canFireAtPlayer
    - Otherwise check if the destination in the direction it is facing is passable
      * IF it is, move there
      * Otherwise, set the direction the opposite direction

**Kleptobot Class**

* virtual void doSomething();
  + Virtual b/c it needs to be redefined for kleptobots. Pseudocode:
  + If ticks have not been initialized, initialize them
  + If kleptobot is dead, return
  + If ticks>0, dec ticks and return
  + Else if ticks = 0
    - If this is an angry kleptobot, check if it can fire at player then return if it does
  + Check if the kleptobot is standing on an item
    - If it is, there is a 1/10 chance it can collect it
      * If it does, make item invisible, set an actor pointer to point to that item, then play appropriate sound
  + If kleptobot has moved distance before turning
    - Reset distancebeforeturning to random value
    - Call moveinranddir
    - Dec ticks
  + Else if destination in dir is passable
    - Move in that direction, decticks, dec distancebeforeturning
  + Else if kleptobot hits an obstruction
    - Reset distancebeforeturning to random value
    - Call moveinranddir
    - Dec ticks
* virtual void damage(int type);
  + Virtual b/c had issues using actor’s damage function to damage this, so I simply decided to redefine it for kleptobots. Pseudocode:
  + Decrease hp
    - If dead, Checks what type of kleptobot it is, plays appropriate sound, and increases score accordingly
    - Otherwise just play impact sound
* virtual void moveInRandDir();
  + Virtual b/c it needs to be accessed by an Actor pointer. Pseudocode:
  + Check if the space in each direction is passable
  + Create an array of all possible directions
  + If spots in a certain direction are not passable, remove that direction from the array
  + If there are no movable directions, simply set direction to a random direction
    - Otherwise, choose a random value from the array and move in that direction and set that direction.
    - Dec distancebeforeturning

**StudentWorld Class**

**All functions are virtual to avoid any compile errors and I ran out of time to check if functions did not have to be virtual or not.**

* virtual int init();
  + Calls loadaLevel()
  + Dependong on its return value, will return if there is a level error, player wins, or game should continue
* virtual int move();
  + Calls setDisplayText
  + Asks player to do something if alive, otherwise returns player dead
  + For every actor in the game
    - Ask the actor to do something
    - Check if player died after that actor did something, dec lives if yes and return correct value if yes
  + Delete any dead actors after everyone has done something
  + If level is complete, return appropriate value, otherwise continue game
* virtual void cleanUp();
  + Remove actor pointers from list and delete all actors, then delete player
* bool isSpaceEmpty(int x, int y);
  + For every actor
    - Get the x,y values of that actor. If those equal inputs, return false
  + Otherwise all actors visited, return true
* virtual int loadALevel();
  + Create the levelname string, then try and load that level
    - Return appropriate values if level not found or bad format
  + If level loaded correctly
    - Go through every point in the grid, and allocate appropriate actors for each char or no actor for spaces
* virtual int objectType(int type);
  + Receives what type of object, and returns an int classifying object as impassable, damageable, special cases, etc
* virtual int numJewels() const { return m\_jewels; }
  + Simply returns current number of jewels uncollected.
* virtual int getBonus() const { return m\_bonus; }
  + Simply returns the current bonus
* virtual Actor\* getPlayer() { return m\_player; }
  + Simply returns player pointer
* virtual Actor\* getObjectAt(int x, int y, list<Actor \*> &actors);
  + Check if player is at input x and way, then return player pointer if yes
  + For every actor in the list provided
    - Check if actor’s x,y equals input x,y
      * If yes, return pointer to that actor
  + Otherwise, return a nullptr
* virtual Actor\* getObjectWithout(int x, int y, Actor \*p, list<Actor \*> &orig, list<Actor \*> &copy);
  + Makes a copy of the list and removes the input object, then calls getObjectAt for the copy list
* virtual list<Actor \*> actors();
  + Simply returns m\_actors
* virtual bool levelComplete();
  + Checks if player is on top of the exit and exit is open
    - If yes, increases score appropriately and then returns true
  + Otherwise returns false
* virtual bool bulletCollision(Actor \*p, int x, int y, GraphObject::Direction dir);
  + Checks if destination is empty, returns false if it is
  + Otherwise, gets the object at destination
    - Check if bullet hits wall, if yes destroy bullet
      * Returns false for noninteresting collision
    - Checks if bullet is on a passable object
      * If yes, keep checking to see if there is a non-passable object there
        + If yes, damage that object and destroy bullet
        + Return true for interesting collision
    - Checks if bullet hits damageable object
      * IF yes, damage object and destroy bullet
    - Checks if bullet hits a factory
      * IF yes, check if robot is on it
        + If there is a robot, damage it, destroy bullet
        + Return true for interesting collision
      * Destroy bullet
* virtual void setExitOpen() { m\_exitOpen = true; }
  + Sets a bool value to exit open to true.
* virtual void decJewels() { m\_jewels--; }
  + Simply decreases the current number of jewels
* virtual void checkDeadActors();
  + Processes through actor list. If an actor is dead, destroy it.
* virtual void setDisplayText();
  + Create variables for all items displayed in display text
  + Create a stringstream and create it using values provided
  + Set the gamestattext to equal this stringstream
* virtual void addActor(Actor \*p);
  + Pushes the actor pointer to the actors list

2. To the best of my knowledge, all functionality has been completed according to the specification and there are currently no known bugs.

3. Design assumptions:

* Multiple items can be on a single position due to killing a kleptobot on that position

4.

* StudentWorld
  + I tested this class by mainly through the init, move, and cleanup functions after each level ended or a tick ended. Essentially I just made sure that each actor that should do something did something, and my helper functions were used mostly in the other classes.
* Actor
  + I tested this function mostly by just ensuring that all of the values were initialized correctly, and made sure that I could access the derived classes through a pointe to this class.
* Wall
  + This class didn’t require much testing. I just had to make sure that the object did nothing every tick, and could not be passed.
* Hole
  + I tested this class by making sure that it was more or less unpassable by all objects besides bullets and boulders. If a boulder moved on top of a hole, I made sure that both were destroyed and removed from the actors list correctly.
* Exit
  + I tested this class by making sure that it was created invisible and then turned visible after all jewels were collected.
* Bullet
  + I tested this class by making sure all bullet collisions were operating correctly. I checked bullet impacts with Kleptobots, Snarlbots, Players, Boulders, and inanimate objects as well as making sure that bullets did not collide with objects such as goodies and empty spaces. This class was tested mainly by playing through a game and checking the various situations where bullets should collide, should damage an object, etc.
* KBotFactory
  + I tested this class by placing a factory in the middle of a 5x5 region encompassed by holes, and made sure that it would only spawn Kleptobots if there were less than 3 in the region and there were none currently on the factory. I made sure that the factory did not spawn additional kleptobots by placing a single factory surrounded on all sides by holes, and making sure that only one bot was spawned.
* Character
  + This class was mostly tested by implementations of the following classes. Besides this, I simply had to make sure that variables were initialized correctly and the member functions of this class worked for the following classes.
* Boulder
  + I tested this class by making sure that it could be destroyed by bullets (both from the player and enemies) as well as making sure that it could be pushed into a hole and would fill that hole (by destroying both objects).
* Player
  + This class was mainly tested through comparisons of player functionality in my game to the sample provided. I ensured that all key presses could create the correct action such as movement, firing bullets, and ending the level early by hitting esc. I also made sure that the player would be damaged when it is hit by a bullet and could collect items and push boulders correctly.
* Robot
  + I tested this class by making sure that all members were initialized correctly. Otherwise, the most testing involved the following bot classes.
* Snarlbot
  + HorizSnarlbot
  + VertSnarlbot
  + I tested the Snarlbot class by ensuring that the AI for firing at players, moving, etc was correct. I made sure that the Snarlbot only fired at the player if the player was in its line of sight unobstructed by any impassable objects besides holes. I also made sure that the bots could be damaged, could pass through items, etc, fired bullets correctly, and turned around when encountering an obstacle. I also ensured that each bot would only take an action after its ticks reached 0.
* Kleptobot
  + AngryKleptobot
  + I tested this class by making sure that they spawned correctly and completed the correct actions. To do so, I placed a factory in an enclosed area of 5x5 filled with items and made sure that the Kleptobots could collect items, and would move in a straight direction until it should turn or until it encountered an obstacle. I tested its movement in a random direction after encountering an obstacle by commenting out the code for it moving in a random direction after moving a certain distance and seeing if it would try to turn after encountering an obstacle, and made sure that the Kleptobots could collect items and would only collect one item.
* Goodie
  + Jewel
  + RestoreHP
  + ExtraLife
  + Ammo
  + For all goodies, I simply tested by creating a level of goodies and then trying to collect them. I also created a closed area with Kleptobots and tested having them collect items and seeing what would happen if I killed them while they were carrying an item