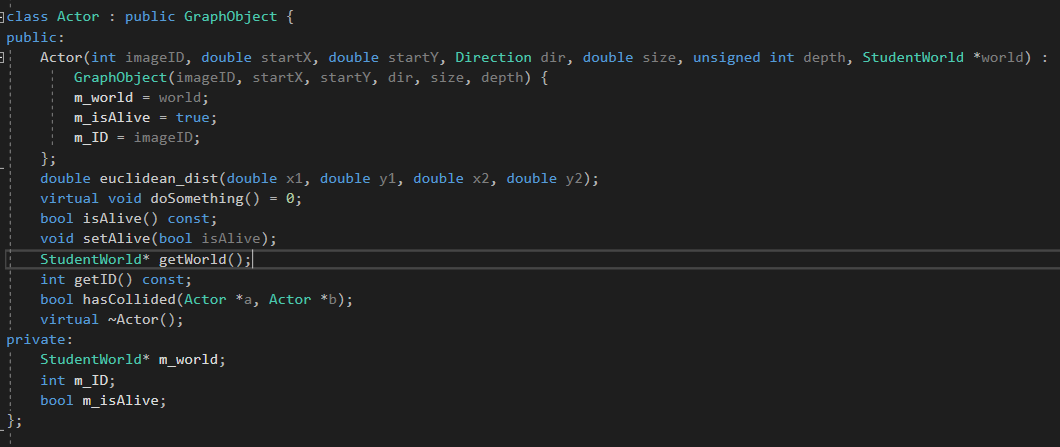
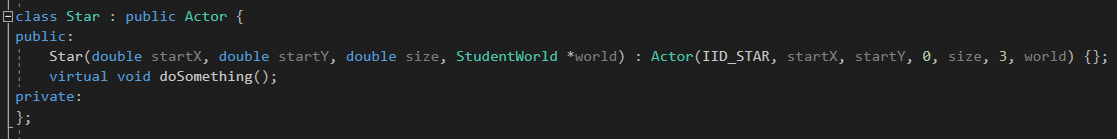
**CLASS ACTOR:**



The Actor class inherits from the GraphObject class that allows us to create physical objects on the screen. The Actor *constructor* takes in several parameters, and initializes the GraphObject class through an initializer list. It also initializes Actor’s member variables. The studentWorld pointer allows us to access member functiosn from StudentWorld and GameWorld. The *Euclidean\_dist()* function takes in 4 doubles, and calculates the Euclidean distance between them (*x* and *y* values of two objects). *doSomething()* is a pure virtual function. This allows us to make the Actor class a virtual base class so we can’t accidentally define an Actor object, and also ensures that every derived class has a *doSomething()* function. The *isAlive()* function is a const member function and returns whether an object is dead or alive. *setAlive()* takes in a boolean and updates the *isAlive()* function through the private member *m\_isAlive*. These functions are defined in the Actor class because all objects need to be set to alive or dead at some point during their doSomething function. *StudentWorld\* getWorld()* allows us to return a pointer to the StudentWorld. This allows us to access StudentWorld and GameWorld classes. *getID()* returns the IID of each object, which is useful when it comes to checking collisions. *hasCollided()* takes in to Actor pointers, thereby using polymorphism and the *Euclidean\_dist* function to determine if two objects have collided. The virtual destructor *~Actor()* ensures that all other destructors can be called with any loss of data.

**Test:** The first test was done in Part 1 when the star class was created. If this class was not setup properly, we wouldn’t be able to display any of the objects on our screen. Indirectly, the function was tested time and again because every other class in this program uses functions from the Actor class, and if it didn’t behave properly, it was modified. Therefore, it was tested by observing if the other derived classes worked correctly.

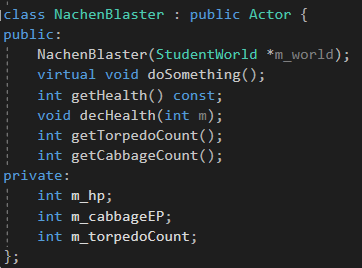
**CLASS STAR:**



The star class inherits from the Actor class and therefore has access to all of its public member functions. The *constructor* takes in 4 parameters, and initializes the Actor class using an initializer list. The *virtual void doSomething* doesn’t need to be virtual since it’s already defined in the base class, but this is believed to be good coding practice. This function moves each start to the left by one pixel and also checks for its boundary conditions so it can invalidate it and delete it from the game by using the *setAlive()* function.

**Test:** The class was tested in part 1. The *doSomething* is called 20 times a second by the *StudentWorld* class. The program was made to run for 30 seconds to a minute to ensure that stars were being generated in the right way. The stars continued to disappear after it reached the left side of the screen, and more starts would appear with random sizes as required by the spec on the right most side of the screen. The class also worked when other classes/objects were created/destroyed.

**CLASS NACHENBLASTER:**

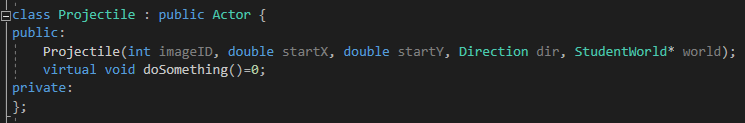


The NachenBlaster class is inherited from the Actor class. This class is needed to create the ship (the NachenBlaster) that travels around the screen. The *constructor* takes in a studentWorld pointer so that the NachenBlaster can be initiated and used by the StudentWorld object. The *virtual void doSomething()* function for the NachenBlaster does several things. Firstly, it checks if the NachenBlaster is alive. It also checks for a user’s keypress and causes the NachenBlaster object to move to the desired location. In the instance that the user presses a space bar, it creates a new Cabbage object. In the instance the user presses the tab key, it creates a torpedo. The *doSomething()* function then goes through all the actors in the actor vector defined in the StudentWorld class and checks for collisions through the hasCollided() function in Actor. These collisions include collisions with projectiles, aliens and goodies. It increases the NachenBlaster’s score accordingly. It is able to check for specific collisions using the getID() function. It compares this IID to specific objects that it is supposed to collide with. If it collides with a turnip or torpedo, the NachenBlaster’s health is decremented and the object is set to dead. The function also checks for collisions with all the three types of Aliens and the three types of goodie using the same method. The *getHealth()* function returns the health of the nachenBlaster, which is needed later in the program to send the NachenBlaster’s health to the string stream. The *decHealth()* function decreases the nachenblaster’s health by the given amount passed in as a parameter. *getTorpedoCount()* returns an int that represents the number of torpedos the nachenBlaster possess at any given point in time. The *getCabbageCount()* function returns the number of cabbages that the nachenBlaster possesses. Most of these functions are normal functions because they only apply to the NachenBlaster class. The *get* Functions help us access all the privates of the NachenBlaster class so that the can be displayed on the string stream at the top of the game as well as help with the general functionality of the NachenBlaster class.

**Test:**

Initially created and checked that it showed up. After creating the projectile class, I made it fire cabbages with a predefined number of cabbages. Then I made it fire torpedos by increasing the count of the torpedos to a limited value. I then checked to see if it took damange when it crashed into an alien ship, and see it eventually died. I also checked if its collision with turnips shot caused damage. Lastly, I checked to make sure the score was being incremented through several cout statements. Once I setup the string stream, I was able to see if the goodie’s benefits were received.

**CLASS PROJECTILE:**

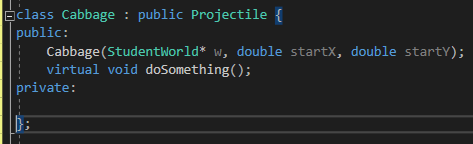


The Projectile class is an abstract base class and is derived from the Actor class. All projectiles are actors, but it’s important to make sure that no ‘Projectile’ object is created. This class acts as a container for all the other projectiles (Cabbages, Turnips and Torpedos). The *constructor()* takes in 5 parameters, including the direction that the projectile travels in. This is important because a Torpedo may have an angle of 0 or 180. The *doSomething()* function is pure virtual to make this a virtual base class, and also ensures that all Projectiles have a *doSomething()* function.

**Test:**

The projectile function is an abstract base class so it could not be checked directly. However, I was able to check if using the derived classes. I did make sure that the Student World function that accept projectile pointers as a parameter to use via polymorphism operate without runtime or compilation errors. This would be added to the Actor\* vector.

**CLASS CABBAGE:**

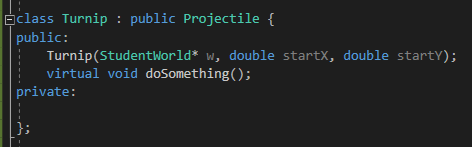


The cabbage class is derived from the Projectile class. The Cabbage *constructor* takes 3 required parameters. The *doSomething()* function checks if the cabbage is not alive, and in this case just returns from the function. It also moves the cabbage 8 pixels to the right and causes it to rotate by 20 degrees every tick. If it goes out of bounds on the right side of the screen, it sets it to dead. These functions are the base essential needed for the Cabbage class, especially because it is derived from Projectile which is derived from Actor.

**Test:**

Ensured that the NachenBlaster object was able to create and shoot cabbages. I slowed down the frame rate to ensure the cabbages rotated in the correct direction. I also ensured through observation that the Cabbages disappeared after they collided with an alien ship, or off the screen if it travelled past the width of the screen.

**CLASS TURNIP:**

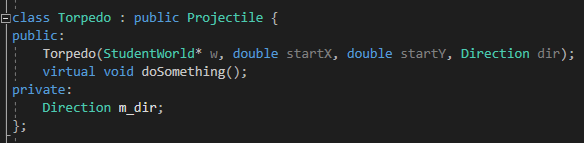


The turnip class is derived from the Projectile class. It’s *constructor()* takes in 3 parameters that are required to setup a Projectile/Actor object. The *void doSomething()* function checks to see if the turnip is alive. If it’s not, it returns from the function. Otherwise it moves to the left by 6 units. If it goes off the screen (), it sets the Turnip’s state to dead. The *doSomething()* function also causes the turnips to rotate at an angle of 20 degrees during every tick.

**Test:**

Ensured that the Alien’s were able to create and shoot turnips. I slowed down the frame rate to ensure the turnips rotated in the correct direction. I also ensured through observation that the turnips disappeared after they collided with the NachenBlaster, or off the screen if it travelled past the width of the screen.

**CLASS TORPEDO:**

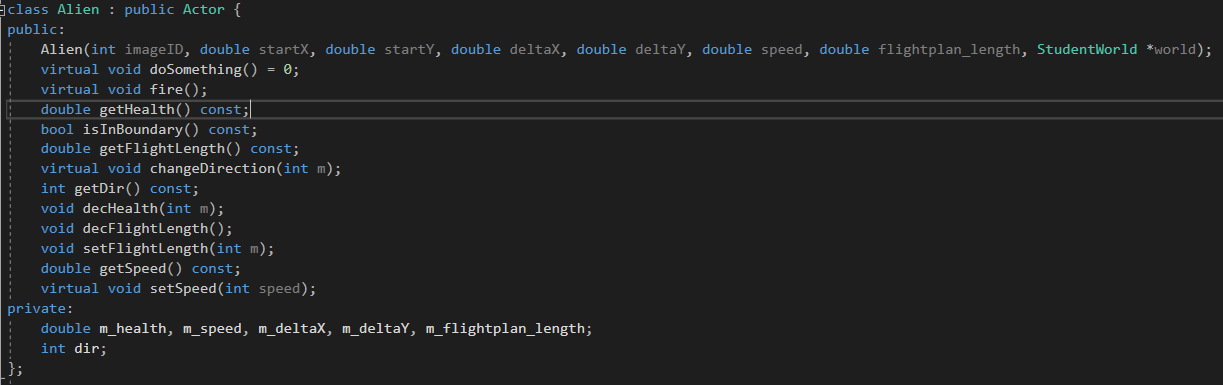


The torpedo class is another derived class and is derived from the Projectile class. The *constructor* for this class is slightly different than the others, because it takes into consideration the direction of the Torpedo. When it is passed into the *constructor* it is also saved as a local variable. This is used in the *void doSomething()* method to determine whether the NachenBlaster or the Snagglegon is firing the projectile. Within the *doSomething()* function, the direction is always taken into consideration while determining projectile behaviour. If the direction is 0, the projectile will move to the right side and will check its boundary conditions for the right side of the screen to determine whether it’s alive or not. If the direction is 180, the projectile will move to the left side and will check its boundary conditions for the left side of the screen to determine whether it’s alive or not. At the beginning of the function, it also checks if the Torpedo is alive or not. If it isn’t, it just returns from the function.

**Test:**

Ensured that the Snagglegon and NachenBlaster were able to create and shoot torpedos. I slowed down the frame rate to ensure the direction was either 0 or 180. I also ensured through observation that the Torpedos disappeared after they collided with the NachenBlaster or Alien, or off the screen if it travelled past the width of the screen in either direction.

**CLASS ALIEN:**

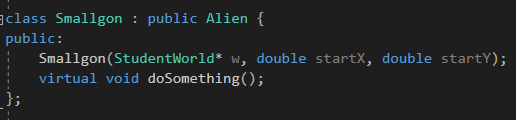


The Alien class is derived from the Actor class. It serves as a container for the three types of aliens (Smallgon, Smoregon and Snagglegon). The *constructor* takes in a long list of parameters, and it initializes each of the private variables to the parameters passed into its *constructor*. The unique parameters here are speed and flightplan\_length. These need to be saved in private members because they’re used to calculate a trajectory for each alien’s movement. The *doSomething()* function is a pure virtual function for two reasons. Firstly, it makes this class an abstract base class so that no object of type Alien can be created. It also mandates that every class derived from Alien needs to have a *doSomething* function. The *fire()* function is a virtual function because it needs to be altered by the Snagglegon since the Snagglegon not only fires a different object, but also has a different probability to determine whether it should fire or not. Presently, the *fire()* function runs through the given probability in the spec and creates turnips if the probability condition is true. Since all aliens from the right side of the screen to the left side of the screen, I created a common boundary function called *isInBoundary()* to see if it has passed the left side of the screen. If it does, it sets the alien’s state to dead. The *getHealth()* function returns the value of the health private member variable. The *getFlightLength()* function is a const function and returns the flight\_length private member. This is needed in the calculation of a possible new flight plan. The *changeDirection()* function takes in an integer and sets the private data member ‘dir’’s value to the given integer. This integer takes the values of 1,2 or 3, indicating travelling left, travelling up and left or travelling down and left. This could be a private member function, but I think it works fine here. It is used to help determine the change in flight plan. The *getDir()* function returns the value of the private member variable. This may be unnecessary, it is used so that all calculations can use function calls rather than raw values. This makes it significantly easier to solve or change if needed. *decHealth()* and *decFlightLength()* decrease their respective member variables. As mentioned earlier, this could be done directly using member variables but it is done through functions for the sake of consistency while doing calculations. *setFlightLength()* updates the value of the flight length as needed while determining the way the aliens will move. The *getSpeed()* and *setSpeed()* update the private member variables as required during the course of calculations.

**Test:**

Since alien is an abstract base class, it was difficult to test it directly since objects of this type could not be created. However, I was able to test it through its derived classes (the different types of aliens). I also made sure that the Student World function that accept alien pointers as a parameter to use via polymorphism operate without runtime or compilation errors.

**CLASS SMALLGON:**

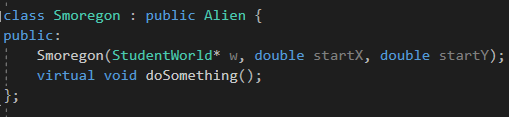


The smallgon class is derived from the Alien class. Its *constructor* takes in 3 parameters, with the startX and startY indicating their start position. The *void doSomething()* is virtual because it changes up the function definition from the Alien class. The *doSomething()* function checks for the Smallgon’s health, and if health is < 0, it sets its state to dead. It also checks if the Smallgon’s boundary conditions are met and sets its state accordingly. It then proceeds to calculate/update the flight path as indicated by the specifications. Lastly, it checks to see if a NachenBlaster fired a cabbage or torpedo. In that case, it decrements the health. If the health of the Smallgon falls to less than 0, it increases the NachenBlaster’s score, creates an explosion object and sets the state of the smallgon to dead.

**Test:**

Checked if the Smallgon object was created by creating a dummy smallgon in my StudentWorld’s init function. I then checked if it took damange from the NachenBlaster’s shooting and head on collision. I also checked to make sure the Smallgon moves as specified, by observation and comparison with the example exe file provided. If also checked to see if they shoot turnips when the NachenBlaster falls in the given range.

**CLASS SMOREGON:**

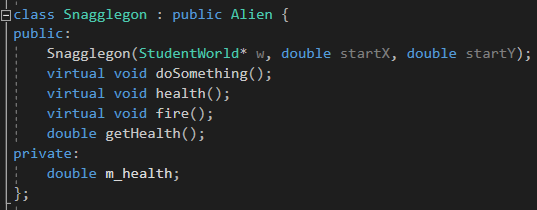


The Smoregon class is derived from the Alien class. Its *constructor* takes in 3 parameters, with the startX and startY indicating their start position. The *void doSomething()* is virtual because it changes up the function definition from the Alien class. The *doSomething()* function checks for the Smoregon’s health, and if health is < 0, it sets its state to dead. It also checks if the Smoregon’s boundary conditions are met and sets its state accordingly. It then proceeds to calculate/update the flight path as indicated by the specifications. In the occasion that a certain probability given in the spec matches up, it allows the Smoregon to speed across the screen at 5 pixels per tick. Lastly, it checks to see if a NachenBlaster fired a cabbage or torpedo. In that case, it decrements the health of the Smoregon. If the health of the Smoregon falls to less than 0, it increases the NachenBlaster’s score, creates an explosion object and sets the state of the Smoregon to dead. It also checks to see if it should drop any of the two required goodies (repair and torpedo), as determined by the spec.

**Test:**

Checked if the Smoregon object was created by creating a dummy Smoregon in my StudentWorld’s init function. I then checked if it took damange from the NachenBlaster’s shooting and head on collision. I also checked to make sure the Smoregon moves as specified, by observation and comparison with the example exe file provided. I also ensured that the Smoregon was able to moving at ramming speed, if also checked to see if they shoot turnips when the NachenBlaster falls in the given range. It also checks to see if Goodies are created when the Smoregon dies.

**CLASS SNAGGLEGON:**

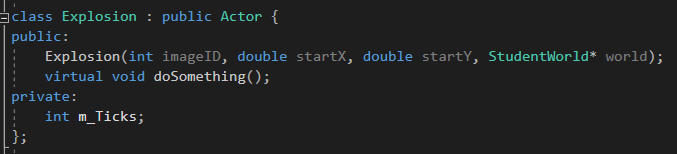


The Snaggelgon class is derived from the Alien class. Its *constructor* takes in 3 parameters, with the startX and startY indicating their start position. The *void doSomething()* is virtual because it changes up the function definition from the Alien class. The *doSomething()* function checks for the Snagglegon’s health, and if health is < 0, it sets its state to dead. It also checks if the Snagglegon’s boundary conditions are met and sets its state accordingly. It then proceeds to calculate/update the flight path as indicated by the specifications. Lastly, it checks to see if a NachenBlaster fired a cabbage or torpedo. In that case, it decrements the health of the Snagglegon. If the health of the Snagglegon falls to less than 0, it increases the NachenBlaster’s score, creates an explosion object and sets the state of the Snagglegon to dead. It also checks to see if it should drop extra life goodie using the probability given in the spec. The Snagglegon’s *health()* function calculates the Snagglegon’s health according to a given formula in the spec and set’s *m\_health* to that value. *getHealth()* allows to return the value of the m\_health private member variable. This is done for consistency’s sake. The *fire()* function is also virtual because it now needs to fire a torpedo with a new probability. This is different from the other two aliens. getHeath() returns the value of the health of the Snagglegon.

**Test:**

Checked if the Snagglegon object was created by creating a dummy Snagglegon in my StudentWorld’s init function. I then checked if it took damage from the NachenBlaster’s shooting and head on collision. I also checked to make sure the Snagglegon moves as specified, by observation and comparison with the example exe file provided. It also checked to see if they shoot Torpedos when the NachenBlaster falls in the given range. It also checks to see if Goodies are created when the Snagglegon dies.

**CLASS EXPLOSION**

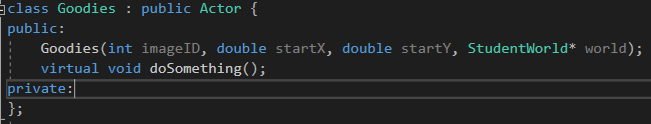


The explosion class is derived from the Actor class. The constructor takes in 4 parameters, so that it can create an explosion object. The *virtual void doSomething()* is a virtual function because the class cannot be created otherwise (since Alien is a virtual base class and doSomething is a pure virtual function). The *doSomething()* checks to see if the explosion object is alive or not. Then it increases the size by 1.5. After this, it increments the number of ticks (m\_Ticks) by 1. If m\_ticks equals 4, it sets the Explosion’s state to dead.

**Test:**

Checked by slowing down the frames per second in the game controller file that the explosion increases in size for 4 ticks and then disappears. Also ensured through observation that the size increases by 1.5 every second.

**CLASS GOODIES:**

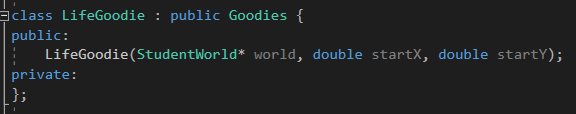


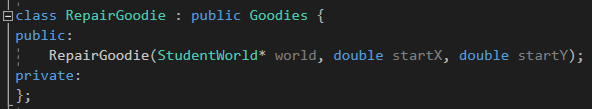
The goodie class is derived from the Actor class. Its *constructor* takes in 4 parameters so that it can be initialized in the current game world. The *virtual void doSomething()* function is not pure virtual, because all the three goodies have the same *doSomething()* function. This function checks if the goodie is alive, and if it is not, it sets its state to dead. It then gets the goodie to move 0.75 pixels down and left and then checks for the boundary condition.

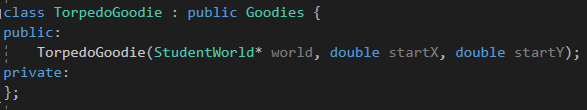
**Test:**

Couldn't check this directly as it is an Abstract Base Class, but made sure that the Student World function that accept goodie pointers as a parameterto use via polymorphism operate without runtime or compilation errors.

**CLASS LIFEGOODIE, REPAIRGOODIE, TORPEDOGOODIE:**







These three classes are derived from the Goodies class. They only contain a *constructor*, because all their other functionality is defined by the base class Goodies. The *constructor* takes in 3 parameters, which allow the Goodie object to be created by the GraphObject class.

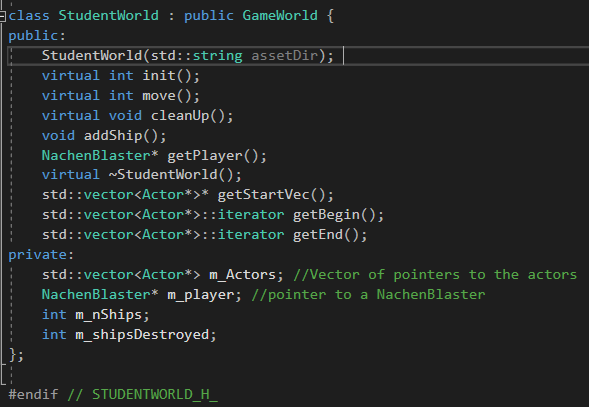
**Test:**

ExtraLifeGoodie: Checked that it is created only if Snagglegon dies and it increases the lives by 1.

FlatulenceGoodie: Checked that it is created only if Smoregon dies and that it increases the Nachenblaster's torpedo count.

RepairGoodie: Checked that it is created only if Smoregon dies and that it increases the Nachenblaster's hp

**CLASS STUDENTWORLD**



The studentWorld class is derived from the GameWorld class. The *init()* function initializes the stars and the NachenBlaster, and returns the integer that tells the game to continue. This creates a playing field when the game is loaded. The *move()* function does several things. It:

* Checks if the player is alive or has zero health and returns GWSTATUS\_PLAYER\_DIED
* Checks if the NachenBlaster is alive. If it is, it calls it doSomething()
* Checks if all the ships are destroyed on a given level
* Updates the stat line
* Counts the number of alien ships destroyed and deletes any dead actors
* Calls the doSomething for all actors
* Creates new stars with a 1/15 probability
* If player isn’t alive anymore, delete it and set the pointer to a nullptr
* Check once again if all the actors are alive or not, and delete any dead actors

The *cleanUp()* function goes through all the remaining actors in the array and deletes them. *Init()*, *move()* and *cleanUp()* are virtual functions because they’re inherited from GameWorld where these functions are pure virtual. The *addShip()* function determines whether you need to add ships or not. The *getPlayer()* returns a pointer to a NachenBlaster object. This is necessary so you can access NachenBlaster’s functions throughout StudentWorld. These are needed in several instances, such as the gameStat line, checking if it’s alive or not, etc. The virtual destructor *~StudentWorld()* function calls the *cleanUp()* function. *getStartVec()* returns a pointer to the first item in the vector of actor pointers. This is helpful since it allows us to access this private member variable through all the other classes using the *getWorld()* function in the Alien class. *getBegin()* returns an iterator to the first element in the m\_Actors vector. *getEnd()* returns an iterator to the last element in the m\_Acotrs vector. These help traverse through the array and check for collisions.

**Test:**

The studentWorld was checked by observing if the other classes work correctly as Student World orchestrates the interactions between all the other classes. All objects were created in the Init() function during testing, which proved that the init() function was working as expected. The move() function works correctly since all objects move as expected. The virtual destructor ensures no memory leaks, and all ships are also created as expected. The NachenBlaster\* getPlayer() function returns a NachenBlaster pointer to an object and this does show up on the screen.

**FUNCTIONALITY:**

I believe all the required functionality has been implemented.

**DESIGN CHOICES:**

One important (and unnecessary) design choice was to use functions to return the privates of almost all my classes. This was to ensure consistency, and to make sure that any other class could access (“see”) the privates of all the other classes. This would allow me to return all the values of any object’s privates at any given point in time. The following additional assumptions and design choices were made.

* Alien Ships are allowed to overlap each other
* Stars cannot and should not overlap
* Goodies are allowed to pass right through aliens
* Alien ships can be created at the same co-ordinates
* It wasn't specified who should check for collisions, so a majority of my collisions are checked by the NachenBlaster object.