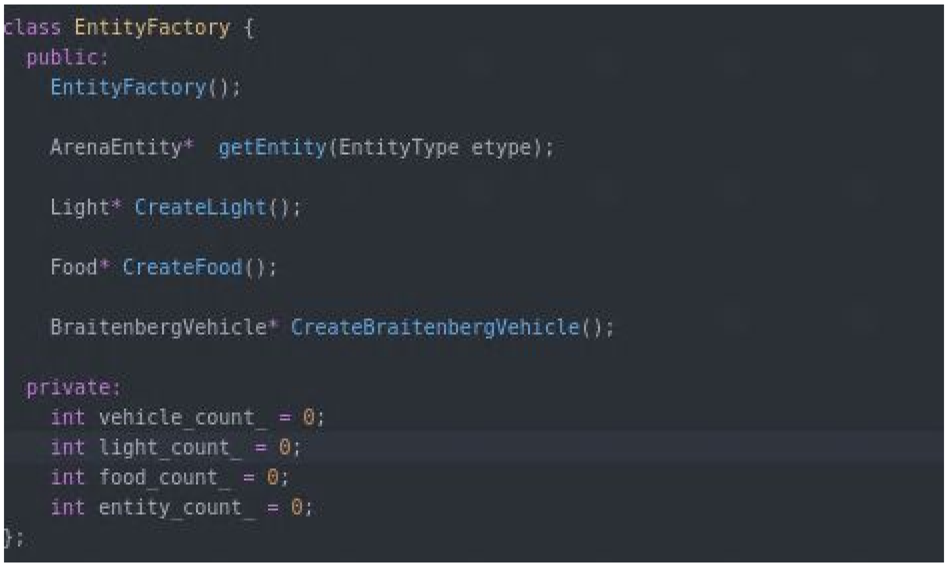
There are 3 versions of the Factory Pattern that is applied to the Braitenberg Vehicle simulation. Those 3 includes:

1. The instantiation of entities in the provided code
2. The use of an abstract Factory class and derived factories as requested in the requirements
3. The use of a concrete Factory class (only 1) that is responsible for the instantiation of all Entity Types

Although instantiating the entities in the provided code is the most simple way to create entities, having a factory proves to be an advantage. By having a factory design pattern this allows the code to become more robust, loosely coupled, and easy to extend. For example, if there is a feature that needs to be added to an entity like food, this can be easily changed because the food class will be unaware of this and this feature can be added to the factory. Also a disadvantage to keep in mind is testing, by instantiating the entity in the provided code it will be difficult to test. A factory would facilitate this by simply returning the entity object and the user does not have to worry about dependency. However there are advantages to instantiating each entity in the provided code. The advantage is that the entity classes instantiate their own objects there for it is independent of association between other entities. Also the organization of the project will be simple due to the fact there is no need to create a factory class and each entity is created in it's own class.

The advantage of creating an abstract factory class and having derived factories is that we have loose coupling. This indicates that each entity (food, light, braitenberg vehicle) have low dependency on each other. Therefore it is easier to test each entity and add new functionality to them. Another advantage of having derived factories is that if there is a specific feature to add to only one entity, it is easily implemented because the entities have low dependence on each other. However, a disadvantage of this is that there are multiple files for each entity therefore the files are somewhat disorganized. Therefore this indicates that this has low cohesion compared to having a concrete factory class. Another disadvantage of this to keep in mind is that when there is a change to the entities overall, one would have to go to each factory and manually edit the code.

The advantage of having one concrete Factory class to produce the Arena entities is that this strategy has high cohesion. This indicates that this concrete Factory for the entities is focused on one objective: creating braitenberg vehicles, lights and food. All of the methods for creating these entities would be stored in this code. *Figure 1* below demonstrates the organizational benefit of having one factory class. Therefore an advantage is the factory is organized. However, the disadvantage of this factory is that this results in high coupling. This indicates that the factory is tightly coupled and changes tend not be non-local, difficult to change and reduces the reuse. Therefore for each entity, braitenberg vehicle, light, and food, they are extremely interdependent. For example, if there is one feature one would like to change on the light entity, this can cause a ripple effect to other entities like food. Another disadvantage is that the code must specify the type of the entity to create from the factory when requesting an entity.



```
class EntityFactory {
  public:
    EntityFactory();

    ArenaEntity*  getEntity(EntityType etype);

    Light* CreateLight();

    Food* CreateFood();

    BraitenbergVehicle* CreateBraitenbergVehicle();

  private:
    int vehicle_count_ = 0;
    int light_count_ = 0;
    int food_count_ = 0;
    int entity_count_ = 0;
};
```

Figure 1: Example of concrete factory class