Jeremy Prater

CS 162 – Assigment 2

Class hierarchy

Class Critter

- Private member critterType (enum)
- Private member step
- Public getter for step -GetStep()
- Public function move()
  - Returns enum direction
- Public function breed()
  - New object gets left in original position
  - We need the last direction moved, so that we know where to put new critter
- public function eat()
  - Increment death counter
  - If doodlebug is eating ant, reset counter
- public function die()
  - Check if step > MaxStep, if so, Kill
- Function Move
- Virtual function GetDeathSteps()
- Virtual function GetBreedSteps()

Class Doodlebug : public Critter

- Max life of 3 steps without eating (doodlebug eats ant)
- critterType = doodlebug
- GetDeathSteps
  - Return 3
- GetBreedSteps
  - Return 8

Class Ant : public Critter

- Max life of 10 steps without eating (Ants eat anything)
- critterType = ant
- GetDeathSteps
  - Return 10

- GetBreedSteps
    - Return 3

The mainGrid will consist of a critter[20][20]

The main loop will iterate through critter[][] and do the following

```
Check if critter already moved this turn.
Get direction from mainGrid[x][y]->move()
Check bounds for move [0,0]-[gridX, gridY]
Check if (newX, newY)
mainGrid[x][y]->eat()
if mainGrid[x][y]->die()
     remove object from grid
if mainGrid[x][y]->breed()
     create new object at oldX, oldY
else
     gridMain (oldX,oldY) = 0
```

Testing

- Verify DoodleBugs die after 3 turns after eating/creation
- Verify DoodleBugs breed every 8 turns
- Verify Ants eat everything
    - Including other ants
    - Ants do not get a timer reset
- Verify DoodleBugs only eat ants
- Verify Critters can not go out of bounds

- Verify all critters are deleted/freed
    - No memory leaks with Valgrind
- Verify no object pointers are over written (lost)

Reflections

Determining if an object had already moved was a problem in this application. I had used a global step counter and ensured that a critter was only allowed to move if the object step counter and the global step counter were in sync. Once an object moved, it's step counter would be greater than the global step counter. However, this introduced issues when breeding because the local step counter for those critters was 0 and it would never be in sync with the global counter.

The solution for this was trivial. The first time an object moves per turn, set a Boolean in its class that indicates it already moved. If object moves down or right, it will be processed again, but the hasMoved variable is true, and the loop will ignore it. The hasMoved member is private and it has public get and set functions.

The other problem I noticed was trying different numbers of death/breed settings to get a self-sustaining equilibrium between ants and doodlebugs. The command line arguments allow overrides:

--doodledeath <n>

--doodlebreed <n>

--antdeath <n>

--antbreed <n>

Another feature that I wanted to implement was random initial conditions. The default uses a random number generator seed of 0, so every run will be identical. I added a run time argument call --random | -r which will seed the random number generator with the system time (msec since 1970).

--random