CSCE A211
Homework #5
50 points total
Due Monday, 11/27, by 11:59 PM

All your programs should use separate files for header and implementation code.

1) 10 pts. Polymorphism and Animation

Starting with Lab 11 as a framework, modify the program so that a Shape class is animated on the screen instead of a Ball class. This will require moving the x/y coordinates and related code from the Ball class into the Shape class. Make the Ball class derived from the Shape class. (It would be more appropriate to rename the Ball class to Circle, but you can skip this unless you feel so inclined).

Add a Square class that is also derived from Shape. Its constructor should take a number that is the length in pixels of one side of the square. The draw function should draw a white square with the center at the x/y coordinates inherited from Shape.

Modify the main function so it creates an array with several squares, balls, and multi-colored balls. They should be visibly animated on the screen.

In summary, the Shape class should be the direct parent of Ball and Square. The Ball class should be the direct parent of MultiColorBall. Make sure that you don't redefine code in a child class that should be inherited from a parent.

2) 25 pts. Doodlebug Simulation

The goal for this programming project is to create a simple 2D predator-prey simulation. Ecologists use simulations like this to study the population dynamics of organisms. In this simulation the prey are ants and the predators are doodlebugs. These critters live in a world composed of a 20x20 grid of cells. Only one critter may occupy a cell at a time. The grid is enclosed, so a critter is not allowed to move off the edges of the world. Time is simulated in time steps. Each critter performs some action every time step.

The ants behave according to the following rules:

1. Move. Every time step, randomly try to move up, down, left or right. If the neighboring cell in the selected direction is occupied or would move the ant off the grid, then the ant stays in the current cell.
2. Breed. If an ant survives for three time steps, then at the end of the third time step (i.e. after moving) the ant will breed. This is simulated by creating a new ant in an adjacent (up, down, left, or right) cell that is empty. This cell can be selected deterministically (e.g. always check left first, then up, etc.) or randomly. If there is no empty cell available then no breeding occurs. Once an offspring is produced an ant cannot produce an offspring until three more time steps have elapsed.

The doodlebugs behave according to the following rules:

1. Move. Every time step, if there is an adjacent ant (up, down, left, or right) then the doodlebug will move to that cell and eat the ant. Otherwise the doodlebug moves according to the same rules as the ant. Note that a doodlebug cannot eat other doodlebugs.
2. Breed. If a doodlebug survives for eight time steps, then at the end of the eighth time step it will spawn off a new doodlebug in the same manner as the ant.

3. Starve. If a doodlebug has not eaten an ant within the last three time steps, then at the end of the third time step it will starve and die. The doodlebug should then be removed from the grid of cells. Note that ants don't starve, they only die by being eaten.

During one turn, all the doodlebugs should move/breed/starve before the ants move/breed.

Write a program to implement this simulation and draw the world using ASCII characters of "o" for an ant and "X" for a doodlebug. Create a class named `Organism` that implements behaviors common to both ants and doodlebugs. For example, this class should have a virtual function named `move` that is defined in the derived classes of `Ant` and `Doodlebug`. You would probably want a virtual function for breeding since that is also similar among both ants and doodlebugs, and perhaps one for starving as well. There may be others depending on your design!

The world should be represented in a class named `World`. This class should contain a 20x20 array of pointers to `Organism` objects. In other words, it should have a class variable: `Organism* grid[SIZE][SIZE]` where `SIZE` is 20. If a cell is cempty, the Organism pointer should be `nullptr`. Otherwise, it should point to an `Ant` or `Doodlebug` object. The `World` class should handle the main logic of the simulation by iterating through the cells and invoking the proper move, breed, or starve functions for the ant or doodlebug referenced by the cell. You may need additional data structures to keep track of which critters have moved.

Before you code, think carefully how to communicate between the world, ant, and doodlebug objects. For example, an ant might need to be able to access the world to know about cells around it. A doodlebug might need to change its position in the world. This will require appropriate communication and/or access between these objects. It may help to diagram your design before you start coding.

Initialize the world with 5 doodlebugs and 100 ants in random locations. After each time step prompt the user to type a key and/or hit enter to move to the next time step. You should see a cyclical pattern between the population of predators and prey, although random perturbations may lead to the elimination of one or both species.

3) 15 pts. Linked List

In Lab 10 we made a `DynamicIntArray` class that allocated a dynamic array of ints based on the number of items in the array. Make a similar class named `IntLinkedList` that has the exact same methods and operators as the completed Lab 10, but uses a linked list of int nodes to store the data.

The behavior of the sample test code should be the same. For example, instead of:

```
DynamicIntArray a;
```

We should be able to replace it with:

```
IntLinkedList a;
```

and the rest of the test code should run and produce the same output, except internally a linked list is used instead of a dynamic int array. Note that the implementation of [] will not be that efficient because you will need to traverse the linked list from the head pointer to reach the desired element.