CSCE 211 Spring 2019

Assignment 5
Due Sunday 4/14, 11:59 PM

1. 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 for ants and doodlebugs (e.g. "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 empty, 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. You will likely need to use a forward class declaration between the World and Organism classes if they are going to store a reference variable to each other. The forward declaration prevents circular dependencies.

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.

2. Polymorphism in C++

This problem has to be completed on transfomer (uaa-transformer.duckdns.org) or on a Unix machine. Log into transformer and copy the files that are
in /home/faculty/kjmock/CSCE_A211/clock to somewhere in your home directory, e.g.:

cp ~kjmock/CSCE_A211/clock/* ~

to copy to your home directory.

There are three files: clock.h, clock.cpp, and main.cpp.

clock.h and clock.cpp contain the header and implementation of a Clock class. If you compile and run the program then it outputs the time in 24 hour format once every second. You have to hit control-c to stop the program from running.

To do:

Write a class named "Alarm" that is derived from the Clock class.

- Your Alarm class must be split up into a separate header and implementation file.
- Make a default constructor that calls the Clock class default constructor.
- Make a constructor that takes the hour, minute, and second for the starting time in addition to an hour, minute and second for an alarm time. It should store the alarm time inside the class and invoke the Clock constructor for the hour, minute, and second starting time.
- Override the tick() virtual function so that it calls the Clock tick() method and then if the current time equals the alarm time, output a message that the alarm has gone off.

Change the main function so it creates an Alarm object and sets the alarm to something close to the starting time. For example, if the starting time is 23 hours, 59 minutes, and 50 seconds, and the alarm time is 0 hours, 0 minutes, and 5 seconds, then the output should look something like this.

```
23:59:50
23:59:51
23:59:52
23:59:53
23:59:54
23:59:55
23:59:56
23:59:57
23:59:58
23:59:59
00:00:00
00:00:01
00:00:02
00:00:03
00:00:04
The alarm went off!
00:00:05
```

You shouldn't need to change anything in the clock class to complete this problem.