CS 3005: Programming in C++

Predator/Prey Simulation Part 1

This assignment is the first of two assignments in which you will construct a simulation program to observe the effects of balance in a predator/prey food chain. We are concerned with two types of Critter's: Lion's (which are predators) and Zebra's (which are prey).

To represent the simulated system, we will use a two dimensional grid of squares, with zero or one critter per square. Each critter is located in one square. It may move to a horizontally or vertically adjacent square. No diagonal moves are allowed.

Critters may die from starvation, or from being eaten by another critter.

All critters can reproduce periodically, with the offspring spawning in an adjacent square.

Assignment

Create the classes Critter, Lion and Zebra, following the programming requirements listed below. You are required to pass the unit tests in CodeGrinder and create some unit tests. [UML Class Diagram]

Critter is a base class used to represent the common data members and methods shared by predators and prey. It also has some abstract methods that must be overridden by its child classes.

Zebras mainly move around and reproduce. There is grass everywhere, so they don’t have to worry about eating.

Lion's have to worry about how many consecutive meals they have missed. If they miss too many, they will die. Zebras are tasty treats. Lion's move around and reproduce as well.

Programming Requirements

Your Critter class must store the following data.

- Two integer values for x and y position of the critter.
- An integer value for the level of the critter in the food chain.
- A boolean value for if the critter is alive.
- An integer value for the number of steps since the critter reproduced.

Your Critter class must have the following methods.

- Critter( int x, int y, int level ); Sets the data members correctly. All critters are created alive. Allows any non-negative value for x and y. Any values less than 0 should be replaced with 0.
- virtual ~Critter( ); The destructor is required, but has an empty block of code for its body.
- int getX( ) const; Returns the x position of the critter.
- int getY( ) const; Returns the y position of the critter.
- int getFoodChainLevel( ) const; Returns the food chain level.
- bool isAlive( ) const; Returns the alive status of the critter.
- int getStepsSinceReproduced( ) const; Returns the number of steps since reproduced.
- bool kill( ); If the critter is alive, make it not alive. Return true if the critter was alive and is now not alive. Otherwise, return false.
- void setPosition( int x, int y ); Sets the position of the critter. Assumes x and y are allowed, does not verify them.
- bool positionAvailable( int x, int y, std::vector< Critter* >& critters, int width, int height ); Returns true if position x,y is a legal position and unoccupied. Otherwise, returns false. Legal positions must have x at least 0 and less than width and y at least 0 and less than height.
- virtual bool move( std::vector< Critter* >& critters, int width, int height ); From the list of empty neighboring locations, randomly chooses one, and moves to it. If no locations are empty, do not move. Do not move to a negative x or y position. Do not move to |x = width| or |y = height|. Returns true if moved. Otherwise, returns false. Dead critters should never move.
- virtual bool eat( std::vector< Critter* >& critters ) = 0; Abstract method, each critter type eats in its own way.
- virtual bool reproduce( std::vector< Critter* >& critters, int width, int height ) = 0;
Abstract method, each critter type reproduces in its own way.

Your `Lion` class must have the following methods.

- `Lion(int x, int y);` All `Lion`s have a food chain level of 10. Initialize the consecutively missed meal count to 0.
- `virtual ~Lion();` Required, with empty code block for body.
- `int getMissedMealCount() const;` Returns the number of consecutive missed meals.
- `Critter *findNeighborPrey(std::vector<Critter*>& critters) const;` Find the first critter in the vector that is alive, has a lower food chain level than the lion, and is next to the lion, either vertically or horizontally. If no such critter is found, return the null pointer (0).
- `virtual bool eat(std::vector<Critter*>& critters);` If no `Zebra`s are nearby, the `Lion` will miss a meal. If the `Lion` has missed 3 or more meals, then the `Lion` dies. If there is a `Zebra` next to the Lion, then the `Lion` moves to the location of the `Zebra`, the `Lion` will eat the `Zebra` and the `Zebra` dies. This should reset the number of consecutively missed meals to 0. Returns true if the `Lion` eats, otherwise returns false. Dead critters should never eat.
- `virtual bool reproduce(std::vector<Critter*>& critters, int width, int height);` Reproduces if there have been at least 8 steps since reproduction and there is an empty adjacent location. Reproduces by creating a new `Lion`, placed in the empty adjacent location and adding it to the vector of critters. If reproduces, sets `mStepsSinceReproduced` to 0, otherwise, adds one to it. Returns true if reproduced, false if not. Dead critters should never reproduce.

Your `Zebra` class must have the following methods.

- `Zebra(int x, int y);` All `Zebra`s have a food chain level of 5.
- `virtual ~Zebra();` Required, with empty code block for body.
- `virtual bool eat(std::vector<Critter*>& critters);` Always returns false. You’ll need to have a statement: `(void)critters` in the body to remove the compiler warnings. Dead critters should never eat.
- `virtual bool reproduce(std::vector<Critter*>& critters, int width, int height);` Reproduces if there have been at least 3 steps since reproduction, there is an empty adjacent location, and a random coin flip is heads. Reproduces by creating a new `Zebra`, placed in the empty adjacent location and adding it to the vector of critters. If reproduces, sets `mStepsSinceReproduced` to 0, otherwise, adds one to it. Returns true if reproduced, false if not. Dead critters should never reproduce.

You must create unit tests for the `reproduce` methods of both `Lion` and `Zebra`.

**Show Off Your Work**

You must pass all of the unit tests in CodeGrinder, plus download them and add to them the unit tests for `reproduce`.

To receive credit for this assignment, you must upload the source code (.cpp and .h files) and the Makefile to the Canvas submission system. This should include `critter.h`, `critter.cpp`, `lion.h`, `lion.cpp`, `zebra.h`, `zebra.cpp`, `Makefile`, and all of the unit test files. The `Makefile` must build a program named `unit-test` that can be run to execute all of your unit tests.

You may assume that `gtest` has been installed on the system used to check your code. Or, you may use your own version of Google Test.