CSE 241 Programming Assignment 4

DUE

May 14, 2025, 23:55

Description

In this PA, you are going to refer to the code provided at the end of this document. Instead of `ant` and `doodlebug`, you are going to have robots.

We have 5 different types of robots: `optimusprime`, `robocop`, `roomba`, `bulldozer`, `kamikaze`. To represent one of these robots we might define a `Robot` class as follows:

Some of the members are given the others are left to you so that you can decide. Decide which of the members are going to be `private` or `public`.

```
class Robot
{

    //a member data which defines the type
    //a member data which stores the strength
    //a member data which stores the hitpoints
    //a helper function which returns the robot type
    Robot( );
    Robot(int newType, int newStrength, int newHit, string name);
    // Initialize robot to new type, strength, hit points
    // Also add appropriate accessor and mutator functions
    // for type, strength, and hit points
    int getDamage();
    // Returns amount of damage this robot
    // inflicts in one round of combat
};
```

Here is an implementation of the `getType( )` function: **We are not going to use this function. Instead we will define this function as `virtual` and provide different implementations for different type of robots.**

```
string Robot::getType()
{
    switch (type)
    {
        case 0: return "optimusprime";
        case 1: return "robocop";
        case 2: return "roomba";
        case 3: return "bulldozer";
        case 4: return "kamikaze";
    }
    return "unknown";
}
```

The `getDamage( )` function outputs and returns the damage this robot can inflict in one round of combat. The rules for calculating the damage are as follows:

- Except a `kamikaze` robot, every robot inflicts damage that is a random number `r`, where `0 < r <= strength`. A `kamikaze` robot inflicts a damage point which is equal to its hitpoints. After inflicting a damage to another robot, the `kamikaze` robot dies.
- `humanic` robots have a `15%` chance of inflicting a tactical nuke attack which is an additional `40` damage points. `optimusprime` and `robocop` are `humanic`.
- With a `20%` chance, `optimusprime` robots inflict a strong attack that doubles the normal amount of damage.
- `roomba` robots are very fast, so they get to attack twice.

A skeleton of `getDamage( )` is given below:

```
int Robot::getDamage()
{
    int damage;
    // All robots inflict damage
    damage = (rand() % strength) + 1;
    //if the robot is a kamikaze robot, the damage is the hitpoints of the kamikaze.
    cout << getType() << " attacks for " <<
    damage << " points!" << endl;
   //calculate additional damage here depending on the type


   //
    return damage;
}
```

Read the following discussion carefully. You have to change the given sample code above. You have to use late-binding instead.

One problem with this implementation is that it is unwieldy to add new robots. Rewrite the class to use inheritance, which will eliminate the need for the variable type. The `Robot` class should be the base class. The classes `kamikaze`, `bulldozer`, `roomba`, and `humanic` should be derived from `Robot`. The classes `optimusprime` and `robocop` should be derived from `humanic`. You will need to rewrite the `getType( )` and `getDamage( )` functions so they are appropriate for each class. For example, the `getDamage( )` function in each class should only compute the damage appropriate for that object. The total damage is then calculated by combining the results of `getDamage( )` at each level of the inheritance hierarchy. As an example, invoking `getDamage( )` for a `optimusprime` object should invoke `getDamage( )` for the `humanic` object which should invoke `getDamage( )` for the `Robot` object. This will compute the basic damage that all robots inflict, followed by the random `15%` damage that `humanic` robots inflict, followed by the double damage that `optimusprime` inflict. Also include mutator and accessor functions for the private variables.

Setup

We are going to have a grid of `Robot` pointers. Then we create robots and randomly place them in the cells of the grid.

- `grid_size`: 10x10
- `initial_count_of_each_robot_type`: 5

Create names for each robot according to the following format:

- `name`: `<type_name_of_the_robot>_<creation_sequence_number_for_each_type>`
- example: `robocop_0`
- `creation_sequence_number_for_each_type` starts from `0` and incremented.
- so, initially you will have `robocop_0, robocop_1,...robocop_5`, `bulldozer_0, bulldozer_1...bulldozer_5`, etc...

Initial values for each robot type is as follows:

- `optimusprime`: strength:100, hitpoints:100
- `robocop`: strength:30, hitpoints:40
- `roomba` strength:3, hitpoints:10
- `bulldozer` strength:50, hitpoints:200
- `kamikaze` strength:10, hitpoints:10

Simulation

Repeat until only one of the robots survive:

- Visit every cell of the grid. If the cell is occupied by robot `R`:
  - `R` tries to move up, down, left or right.

- If the movement direction is occupied by another robot, `R` fights with that robot until one of them is dead. (`the fight loop`)
- **If the cell is empty, `R` moves to that location and keeps moving until it hits another robot.**
- Every robot has one chance of fight for every step of simulation. (you have to keep a flag in every robot and skip the robot if it is already moved. you have reset the flags of every object before the next scan(the step of the simulation) of the grid.)

the fight loop

Lets say, robot R(attacker) tries to fight with robot S(victim). Here is the algorithm:

Repeat until R or S dies:

- `R` calls `getDamage()`. `getDamage()` returns `d_r`.
- `hitpoints` of `S` is decremented by `d_r`.
- print `hit_message`(see `hit_message` for details)
- If `S` is dead, return.
- `S` calls `getDamage()`. `getDamage()` returns `d_s`.
- `hitpoints` of `R` is decremented by `d_s`.
- print `hit_message`(see `hit_message` for details)

`hit_message`

The hit message has two lines. The format of hit message is as follows:

- `<name_attacker>(<hitpoits_attacker>) hits <name_of_the_victim>(<hitpoints_victim_before_hit>) with <damage_inflicted>`
- `The new hitpoints of <name_victim> is <hitpoins_victim>`
- Example:
- `roomba_1(10) hits robocop_4(10) with 3`
- `The new hitpoints of robocop_4 is 7`

Death of a robot

If the `hitpoints` is less than or equal to `0`, the robot is announced as dead. Dead robots should be removed from the grid.

Healing

Healing is the gradual increase of hitpoints. Only the humanic robots can heal. Healing stops during combat. (fight loop). Humanic robots gain 1 hit points at each simulation step.

Turn In

- A zip file containing all the `.cpp` and `.h` files of your implementation. Properly name your files according to the classes you your. Put your driver program(main function) in `main.cpp`.
- Create a simple `MAKEFILE` for your submission. (You can find tutorials for creating a simple make file.)
- Name of the file should be in this format: `<full_name>_PA4.zip`. Don't send `.rar` or `.7z` or any other format. Properly create a `.zip` file from your source files.
- You don't need to use an IDE for this assignment. Your code will be compiled and run in a command window.
- Your code will be compiled and tested on a Linux machine(Ubuntu). GCC will be used.
- Make sure you don't get compile errors when you issue this command : `g++ -std=c++11 <any_of_your_files>.cpp`.
- Makes sure you don't get link errors.

Late Submission

- Not accepted

Grading (Tentative)

- `Max Grade` : `100`.
- Multiple tests will be performed.

All of the followings are possible deductions from `Max Grade`.

- Do **NOT** use hard-coded values. If you use you will loose `10pts`.
- No submission: `-100`.
- Compile errors: `-100`.
- Irrelevant code: `-100`.
- Major parts are missing: `-100`.
- Unnecessarily long code: `-30`.
- Inefficient implementation: `-20`.
- Using language elements and libraries which are not allowed: `-100`.
- Not caring about the structure and efficiency: `-30`. (avoid using hard-coded values, avoid hard-to-follow expressions, avoid code repetition, avoid unnecessary loops).
- Significant number of compiler warnings: `-10`.
- Not commented enough: `-10`. (Comments are in English. Turkish comments are not accepted).
- Source code encoding is not `UTF-8` and characters are not properly displayed: `-5`. (You can use 'Visual Studio Code', 'Sublime Text', 'Atom' etc... Check the character encoding of your text editor and set it to `UTF-8`).
- **Not using virtual functions `-100`.**
- **Not using class inheritance `-100`.**
- **Not de-allocating dynamic memory `-20`.**
- Output format is wrong `-20`.
- Calculation is wrong `-20`.
- Infinite loop: `Fails the test`.
- Segmentation fault: `Fails the test`.
- Fails 5 or more random tests: `-100`.
- Fails the test: `deduction up to 20`.
- Unwanted chars and spaces in output: `-30`.
- Submission includes files other than the expected: `-10`.
- Submission does not follow the file naming convention: `-10`.
- Sharing or inheriting code: `-200`.

The Reference Project and the Code

The goal for this programming project is to create a simple 2D predator-prey simulation. 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 model:

• 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. • Breed. If an ant survives for three time steps, then at the end of the 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. 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 model:

- 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.
- Breed. If a doodlebug survives for eight time steps, then at the end of the time step it will spawn off a new doodlebug in the same manner as the ant.
- 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.

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

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 encapsulates basic data common to both ants and doodlebugs. This class should have a virtual function named move that is defined in the derived classes of Ant and Doodlebug. You may need additional data structures to keep track of which critters have moved

Initialize the world with 5 doodlebugs and 100 ants. After each time step prompt the user to press 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.

```
//simulation.cpp
//
//This program simulates a 2D world with predators and prey.
//The predators (doodlebugs) and prey (ants) inherit from the
//  Organism class that keeps track of basic information about each
//  critter (time ticks since last bred, position in the world).
//
//The 2D world is implemented as a separate class, World,
//  that contains a 2D array of pointers to type Organism.
//
//Normally the classes would be defined in separate files, but
//  they are all included here for ease of use with CodeMate.

#include <iostream>
#include <string>
#include <vector>
#include <cstdlib>
#include <time.h>

using namespace std;

const int WORLDSIZE = 20;
const int INITIALANTS = 100;
```

```
const int INITIALBUGS = 5;
const int DOODLEBUG = 1;
const int ANT = 2;
const int ANTBREED = 3;
const int DOODLEBREED = 8;
const int DOODLESTARVE = 3;

// Forward declaration of Organism classes so we can reference it
// in the World class
class Organism;
class Doodlebug;
class Ant;

// ========================================
// The World class stores data about the world by creating a
// WORLDSIZE by WORLDSIZE array of type Organism.
// NULL indicates an empty spot, otherwise a valid object
// indicates an ant or doodlebug.  To determine which,
// invoke the virtual function getType of Organism that should return
// ANT if the class is of type ant, and DOODLEBUG otherwise.
// ========================================

class World
{
friend class Organism;          // Allow Organism to access grid
friend class Doodlebug;         // Allow Organism to access grid
friend class Ant;           // Allow Organism to access grid
public:
    World();
    ~World();
    Organism* getAt(int x, int y);
    void setAt(int x, int y, Organism *org);
    void Display();
    void SimulateOneStep();
private:
    Organism* grid[WORLDSIZE][WORLDSIZE];
};


// ========================================
// Definition for the Organism base class.
// Each organism has a reference back to
// the World object so it can move itself
// about in the world.
// ========================================
class Organism
{
friend class World;         // Allow world to affect organism
public:
    Organism();
    Organism(World *world, int x, int y);
    ~Organism();
    virtual void breed() = 0;   // Whether or not to breed
    virtual void move() = 0;    // Rules to move the critter
    virtual int getType() =0;   // Return if ant or doodlebug
    virtual bool starve() = 0;  // Determine if organism starves
```

```
protected:
    int x,y;            // Position in the world
    bool moved;         // Bool to indicate if moved this turn
    int breedTicks;         // Number of ticks since breeding
    World *world;
};

// ======================
// World constructor, destructor
// These classes initialize the array and
// releases any classes created when destroyed.
// ======================
World::World()
{
    // Initialize world to empty spaces
    int i,j;
    for (i=0; i<WORLDSIZE; i++)
    {
        for (j=0; j<WORLDSIZE; j++)
        {
            grid[i][j]=NULL;
        }
    }
}

World::~World()
{
    // Release any allocated memory
    int i,j;
    for (i=0; i<WORLDSIZE; i++)
    {
        for (j=0; j<WORLDSIZE; j++)
        {
            if (grid[i][j]!=NULL) delete (grid[i][j]);
        }
    }
}

// ======================
// getAt
// Returns the entry stored in the grid array at x,y
// ======================
Organism* World::getAt(int x, int y)
{
    if ((x>=0) && (x<WORLDSIZE) && (y>=0) && (y<WORLDSIZE))
        return grid[x][y];
    return NULL;
}

// ======================
// setAt
// Sets the entry at x,y to the
// value passed in.  Assumes that
// someone else is keeping track of
// references in case we overwrite something
// that is not NULL (so we don't have a memory leak)
```

```
// ======================
void World::setAt(int x, int y, Organism *org)
{
    if ((x>=0) && (x<WORLDSIZE) && (y>=0) && (y<WORLDSIZE))
    {
        grid[x][y] = org;
    }
}

// ======================
// Display
// Displays the world in ASCII.  Uses o for ant, X for doodlebug.
// ======================
void World::Display()
{
    int i,j;
    cout << endl << endl;
    for (j=0; j<WORLDSIZE; j++)
    {
        for (i=0; i<WORLDSIZE; i++)
        {
            if (grid[i][j]==NULL)
                cout << ".";
            else if (grid[i][j]->getType()==ANT)
                cout << "o";
            else cout << "X";
        }
        cout << endl;
    }
}

// ======================
// SimulateOneStep
// This is the main routine that simulates one turn in the world.
// First, a flag for each organism is used to indicate if it has moved.
// This is because we iterate through the grid starting from the top
// looking for an organism to move . If one moves down, we don't want
// to move it again when we reach it.
// First move doodlebugs, then ants, and if they are still alive then
// we breed them.
// ======================
void World::SimulateOneStep()
{
    int i,j;
    // First reset all organisms to not moved
    for (i=0; i<WORLDSIZE; i++)
        for (j=0; j<WORLDSIZE; j++)
        {
            if (grid[i][j]!=NULL) grid[i][j]->moved = false;
        }
    // Loop through cells in order and move if it's a Doodlebug
    for (i=0; i<WORLDSIZE; i++)
        for (j=0; j<WORLDSIZE; j++)
        {
         if ((grid[i][j]!=NULL) && (grid[i][j]->getType()==DOODLEBUG))
         {
```

```
            if (grid[i][j]->moved == false)
            {
             grid[i][j]->moved = true; // Mark as moved
             grid[i][j]->move();
            }
         }
        }
    // Loop through cells in order and move if it's an Ant
    for (i=0; i<WORLDSIZE; i++)
        for (j=0; j<WORLDSIZE; j++)
        {
            if ((grid[i][j]!=NULL) && (grid[i][j]->getType()==ANT))
            {
                if (grid[i][j]->moved == false)
                {
                 grid[i][j]->moved = true; // Mark as moved
                 grid[i][j]->move();
                }
            }
        }
    // Loop through cells in order and check if we should breed
    for (i=0; i<WORLDSIZE; i++)
        for (j=0; j<WORLDSIZE; j++)
        {
            // Kill off any doodlebugs that haven't eaten recently
            if ((grid[i][j]!=NULL) &&
                (grid[i][j]->getType()==DOODLEBUG))
            {
                if (grid[i][j]->starve())
                {
                    delete (grid[i][j]);
                    grid[i][j] = NULL;
                }
            }
        }
    // Loop through cells in order and check if we should breed
    for (i=0; i<WORLDSIZE; i++)
        for (j=0; j<WORLDSIZE; j++)
        {
            // Only breed organisms that have moved, since
            // breeding places new organisms on the map we
            // don't want to try and breed those
            if ((grid[i][j]!=NULL) && (grid[i][j]->moved==true))
            {
                grid[i][j]->breed();
            }
        }
}

// ======================
// Organism Constructor
// Sets a reference back to the World object.
// ======================
Organism::Organism()
{
    world = NULL;
```

```
    moved = false;
    breedTicks = 0;
    x=0;
    y=0;
}
Organism::Organism(World *wrld, int x, int y)
{
    this->world = wrld;
    moved = false;
    breedTicks = 0;
    this->x=x;
    this->y=y;
    wrld->setAt(x,y,this);
}

// ======================
// Organism destructor
// No need to delete the world reference,
// it will be destroyed elsewhere.
// ======================
Organism::~Organism()
{
}

// --------------------------------
// ----- ENTER YOUR CODE HERE -----
// --------------------------------

// Start with the Ant class
class Ant : public Organism
{
 friend class World;
 public:
    Ant();
    Ant(World *world, int x, int y);
    void breed();   // Must define this since virtual
    void move();    // Must define this since virtual
    int getType();  // Must define this since virtual
    bool starve()   // Return false, ant never starves
      { return false; }
};

// ======================
// Ant constructor
// ======================
Ant::Ant() : Organism()
{
}

Ant::Ant(World *world, int x, int y) : Organism(world,x,y)
{
}

// ======================
// Ant Move
// Look for an empty cell up, right, left, or down and
```

```
// try to move there.
// ======================
void Ant::move()
{
    // Pick random direction to move
    int dir = rand() % 4;
    // Try to move up, if not at an edge and empty spot
    if (dir==0)
    {
        if ((y>0) && (world->getAt(x,y-1)==NULL))
        {
         world->setAt(x,y-1,world->getAt(x,y));  // Move to new spot
         world->setAt(x,y-1,this);  // Move to new spot
         world->setAt(x,y,NULL);
         y--;
        }
    }
    // Try to move down
    else if (dir==1)
    {
        if ((y<WORLDSIZE-1) && (world->getAt(x,y+1)==NULL))
        {
         world->setAt(x,y+1,world->getAt(x,y));  // Move to new spot
         world->setAt(x,y,NULL);  // Set current spot to empty
         y++;
        }
    }
    // Try to move left
    else if (dir==2)
    {
        if ((x>0) && (world->getAt(x-1,y)==NULL))
        {
         world->setAt(x-1,y,world->getAt(x,y));  // Move to new spot
         world->setAt(x,y,NULL);  // Set current spot to empty
         x--;
        }
    }
    // Try to move right
    else
    {
        if ((x<WORLDSIZE-1) && (world->getAt(x+1,y)==NULL))
        {
         world->setAt(x+1,y,world->getAt(x,y));  // Move to new spot
         world->setAt(x,y,NULL);  // Set current spot to empty
         x++;
        }
    }
}

// ======================
// Ant getType
// This virtual function is used so we can determine
// what type of organism we are dealing with.
// ======================
int Ant::getType()
{
```

```
    return ANT;
}

// ======================
// Ant breed
// Increment the tick count for breeding.
// If it equals our threshold, then clone this ant either
// above, right, left, or below the current one.
// ======================
void Ant::breed()
{
    breedTicks++;
    if (breedTicks == ANTBREED)
    {
        breedTicks = 0;
        // Try to make a new ant either above, left, right, or down
        if ((y>0) && (world->getAt(x,y-1)==NULL))
        {
            Ant *newAnt = new Ant(world, x, y-1);
        }
        else if ((y<WORLDSIZE-1) && (world->getAt(x,y+1)==NULL))
        {
            Ant *newAnt = new Ant(world, x, y+1);
        }
        else if ((x>0) && (world->getAt(x-1,y)==NULL))
        {
            Ant *newAnt = new Ant(world, x-1, y);
        }
        else if ((x<WORLDSIZE-1) && (world->getAt(x+1,y)==NULL))
        {
            Ant *newAnt = new Ant(world, x+1, y);
        }
    }
}

// ****************************************************
// Now define Doodlebug Class
// ****************************************************

class Doodlebug : public Organism
{
 friend class World;
 public:
    Doodlebug();
    Doodlebug(World *world, int x, int y);
    void breed();   // Must define this since virtual
    void move();    // Must define this since virtual
    int getType();  // Must define this since virtual
    bool starve();  // Check if a doodlebug starves to death
 private:
    int starveTicks;    // Number of moves before starving
};

// ======================
// Doodlebug constructor
// ======================
```

```
Doodlebug::Doodlebug() : Organism()
{
    starveTicks = 0;
}

Doodlebug::Doodlebug(World *world, int x, int y) : Organism(world,x,y)
{
    starveTicks = 0;
}

// ======================
// Doodlebug move
// Look up, down, left or right for a bug.  If one is found, move there
// and eat it, resetting the starveTicks counter.
// ======================
void Doodlebug::move()
{
    // Look in each direction and if a bug is found move there
    // and eat it.
    if ((y>0) && (world->getAt(x,y-1)!=NULL) &&
            (world->getAt(x,y-1)->getType() == ANT))
    {
        delete (world->grid[x][y-1]);  // Kill ant
        world->grid[x][y-1] = this;    // Move to spot
        world->setAt(x,y,NULL);
        starveTicks =0 ;        // Reset hunger
        y--;
        return;
    }
    else if ((y<WORLDSIZE-1) && (world->getAt(x,y+1)!=NULL) &&
            (world->getAt(x,y+1)->getType() == ANT))
    {
        delete (world->grid[x][y+1]);  // Kill ant
        world->grid[x][y+1] = this;    // Move to spot
        world->setAt(x,y,NULL);
        starveTicks =0 ;        // Reset hunger
        y++;
        return;
    }
    else if ((x>0) && (world->getAt(x-1,y)!=NULL) &&
            (world->getAt(x-1,y)->getType() == ANT))
    {
        delete (world->grid[x-1][y]);  // Kill ant
        world->grid[x-1][y] = this;    // Move to spot
        world->setAt(x,y,NULL);
        starveTicks =0 ;        // Reset hunger
        x--;
        return;
    }
    else if ((x<WORLDSIZE-1) && (world->getAt(x+1,y)!=NULL) &&
            (world->getAt(x+1,y)->getType() == ANT))
    {
        delete (world->grid[x+1][y]);  // Kill ant
        world->grid[x+1][y] = this;    // Move to spot
        world->setAt(x,y,NULL);
        starveTicks =0 ;        // Reset hunger
```

```
        x++;
        return;
    }


    // If we got here, then we didn't find food.  Move
    // to a random spot if we can find one.
    int dir = rand() % 4;
    // Try to move up, if not at an edge and empty spot
    if (dir==0)
    {
        if ((y>0) && (world->getAt(x,y-1)==NULL))
        {
         world->setAt(x,y-1,world->getAt(x,y));  // Move to new spot
         world->setAt(x,y,NULL);
         y--;
        }
    }
    // Try to move down
    else if (dir==1)
    {
        if ((y<WORLDSIZE-1) && (world->getAt(x,y+1)==NULL))
        {
         world->setAt(x,y+1,world->getAt(x,y));  // Move to new spot
         world->setAt(x,y,NULL);  // Set current spot to empty
         y++;
        }
    }
    // Try to move left
    else if (dir==2)
    {
        if ((x>0) && (world->getAt(x-1,y)==NULL))
        {
         world->setAt(x-1,y,world->getAt(x,y));  // Move to new spot
         world->setAt(x,y,NULL);  // Set current spot to empty
         x--;
        }
    }
    // Try to move right
    else
    {
        if ((x<WORLDSIZE-1) && (world->getAt(x+1,y)==NULL))
        {
         world->setAt(x+1,y,world->getAt(x,y));  // Move to new spot
         world->setAt(x,y,NULL);  // Set current spot to empty
         x++;
        }
    }
    starveTicks++;      // Increment starve tick since we didn't
                // eat on this turn
}

// ======================
// Doodlebug getType
// This virtual function is used so we can determine
// what type of organism we are dealing with.
```

```
// ======================
int Doodlebug::getType()
{
    return DOODLEBUG;
}

// ======================
// Doodlebug breed
// Creates a new doodlebug adjacent to the current cell
// if the breedTicks meets the threshold.
// ======================
void Doodlebug::breed()
{
    breedTicks++;
    if (breedTicks == DOODLEBREED)
    {
        breedTicks = 0;
        // Try to make a new ant either above, left, right, or down
        if ((y>0) && (world->getAt(x,y-1)==NULL))
        {
            Doodlebug *newDoodle = new Doodlebug(world, x, y-1);
        }
        else if ((y<WORLDSIZE-1) && (world->getAt(x,y+1)==NULL))
        {
            Doodlebug *newDoodle = new Doodlebug(world, x, y+1);
        }
        else if ((x>0) && (world->getAt(x-1,y)==NULL))
        {
            Doodlebug *newDoodle = new Doodlebug(world, x-1, y);
        }
        else if ((x<WORLDSIZE-1) && (world->getAt(x+1,y)==NULL))
        {
            Doodlebug *newDoodle = new Doodlebug(world, x+1, y);
        }
    }
}

// ======================
// Doodlebug starve
// Returns true or false if a doodlebug should die off
// because it hasn't eaten enough food.
// ======================
bool Doodlebug::starve()
{
    // Starve if no food eaten in last DOODLESTARVE time ticks
    if (starveTicks > DOODLESTARVE)
    {
        return true;
    }
    else
    {
        return false;
    }
}

// ------------------------------
```

```
// --------- END USER CODE --------
// --------------------------------



// ======================
//     main function
// ======================
int main()
{
  string s;
  srand(time(NULL));        // Seed random number generator
  World w;

  // Randomly create 100 ants
  int antcount = 0;
  while (antcount < INITIALANTS)
  {
    int x = rand() % WORLDSIZE;
    int y = rand() % WORLDSIZE;
    if (w.getAt(x,y)==NULL)     // Only put ant in empty spot
    {
        antcount++;
        Ant *a1 = new Ant(&w,x,y);
    }
  }
  // Randomly create 5 doodlebugs
  int doodlecount = 0;
  while (doodlecount < INITIALBUGS)
  {
    int x = rand() % WORLDSIZE;
    int y = rand() % WORLDSIZE;
    if (w.getAt(x,y)==NULL)     // Only put doodlebug in empty spot
    {
        doodlecount++;
        Doodlebug *d1 = new Doodlebug(&w,x,y);
    }
  }

  // Run simulation forever, until user cancels
  while (true)
  {
    w.Display();
    w.SimulateOneStep();
    cout << endl << "Press enter for next step" << endl;
    getline(cin,s);
  }
  return 0;
}
```