```cpp

#include <stdio.h>
#include <iostream>
#include <vector>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <string>
#include <algorithm>
#include <Windows.h>


// constants for use in final conditions
#define ROW 30
#define COL 30
#define HOB 150
#define NAZ 10

using namespace std;

class Cell {

    protected:
        // values - these should be protected
        char name;
        bool moved = true;
        bool breed = false; // helps reset age
        bool dead = false;
        int age = 0;

        /*
        returns all the cell numbers next to the current cell, warping as needed to allow default move
        */
        vector<int> posMoves(int loc) {
            vector<int> moves;

            // up = loc - COL, if less then 0, then correct
            int up = loc - COL;
            if (up < 0) { up = (ROW * COL) + up; } // if up is less then zero, then we want to add the negative
            moves.push_back(up);

            // down = loc + COL, if more than or equal to
            int dow = loc + COL;
            if (dow >= (ROW * COL)) { dow = dow - (ROW * COL); }
            moves.push_back(dow);

            // left: -1 unless left is on the left side
            int le;
            if ((loc % COL) == 0) { le = loc + (COL - 1); }
            else { le = loc - 1; }
            moves.push_back(le);

            // right: +1 unless right is on the right side
            int ri;
            if ((loc % COL) == (COL - 1)) { ri = loc - (COL - 1); }
            else { ri = loc + 1; }
            moves.push_back(ri);

            random_shuffle(moves.begin(), moves.end());
            return moves;
        };

        /*
        narrows down a list of posMoves (or a catered list) to spots on the grid that match
        the char type returns the move or -1
        */
        int singleMove(vector<int> moves, vector<Cell*> grid, char type) {
            for (int i = 0; i < moves.size(); i++) {
                if (grid[moves[i]]->getName() == type) {
                    return moves[i];
                }
            }
            return -1;
        }

        /*
        increases the age of the living, or resets age to 0 and seets breed to true
        */
        void ageUp(int max) {
            if (age != max) {
                age ++;
            }
            else {
                breed = true;
                age = 0;
            }
        }

    public:

        Cell() { name = '.'; }

        char getName() {
            return name;
```

```cpp
        }
        bool getMoved() {
            return moved;
        }
        bool getBreed() {
            return breed;
        }
        bool getDead() {
            return dead;
        }
        int getAge() {
            return age;
        }

        void setBreed(bool bre) {
            breed = bre;
        }
        void setMoved(bool mov) {
            moved = mov;
        }


        virtual int move(vector<Cell*> grid, int loc) {
            return -1;
        }

        virtual int breeding(vector<Cell*> grid, int loc) {
            return -1;
        }

};

class Hobbit : public Cell {
    public:
        Hobbit() : Cell() {
            name = 'O';
        };

        int move(vector<Cell*> grid, int loc) {

            // check to see if the age is 3. (the extra +1 due to calling x2)
            ageUp(3);

            vector<int> moves = posMoves(loc); // the possable moves
            return singleMove(moves, grid, '.');
        }

        int breeding(vector<Cell*> grid, int loc) {
            vector<int> open = posMoves(loc); // the possable moves
            return singleMove(open, grid, '.');
        }
};

class Nazghul : public Cell {
    private:
        int lastFeed = 0; // how long from the last time it was fed


        /*
        clears the teleport moves off the move list, as the nazghuls can not teleport
        */
        vector<int> cleanMove(vector<int> rawMoves, int loc) {
            vector<int> moves;

            for (int i = 0; i < rawMoves.size(); i++) {
                if ((rawMoves[i] == (loc - 1)) || (rawMoves[i] == (loc - COL)) ||
                    (rawMoves[i] == (loc + 1)) || (rawMoves[i] == (loc + COL))) {
                    moves.push_back(rawMoves[i]);
                }
            }
            return moves;
        }

    public:
        Nazghul() : Cell() {
            name = 'X';
        };


        int move(vector<Cell*> grid, int loc) {

            // track moved
            // track age like with hobbit 0 - 8 inclusive
            ageUp(8);

            vector<int> moves = posMoves(loc); // for cleaning up options not okay for nazzys
            moves = cleanMove(moves, loc); // where the final moves will go.
            int move; // to hold the testable move

            move = singleMove(moves, grid, 'O');
            if (move != -1) {
                lastFeed = 0;
                return move;
            }

            lastFeed++;
            if (lastFeed == 3) { // per spec, above move
                dead = true;
            }

            // while already determined dead, this is to allow it to
            // emulate one last desperate search for food.
            return singleMove(moves, grid, '.');
        }

        int breeding(vector<Cell*> grid, int loc) {
            vector<int> open = posMoves(loc); // the locs
            open = cleanMove(open, loc);
            int spot;

            spot = singleMove(open, grid, 'O');
            if (spot != -1) {
                return spot;
            }
            return singleMove(open, grid, '.');
        }
};

/*
handler f(x)n for color change in console
*/
void changeColor(int color) {
    HANDLE hConsole;

    hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(hConsole, color);
}

/*
a 1d vector map. Map was made as a 1d vector to simplfiy project
*/
vector<Cell*> makeMap() {
    vector<Cell*> grid;

    for (int i = 0; i < ROW; i++) {
        for (int j = 0; j < COL; j++) {

            Cell *cell = new Cell();
            grid.push_back(cell);
        }
    }
```

```cpp

    return grid;
}

/*
seeds the map
*/
vector<Cell*> initalConditions(vector<Cell*>& grid) {
    vector<int> unique;
    int counter = 0;

    // make a list of unique indexs
    while (counter < (HOB + NAZ)) {

        int testIndex = rand() % (ROW * COL); // random number in range of grid
        bool used = false; // is the index already taken?

        for (int i = 0; i < counter; i++) {
            if (unique[i] == testIndex) { used = true; } // if used is true, the number is garbage
        }
        if (used == false) {
            unique.push_back(testIndex);
            counter ++;
        }
    }

    // use constants from above
    for (int i = 0; i < unique.size(); i++) {
        int loc = unique[i];
        if (i < NAZ) {
            Nazghul *naz = new Nazghul();
            grid[loc] = naz;
        }
        else {
            Hobbit *hob = new Hobbit();
            grid[loc] = hob;
        }
    }

    return grid;
}

/*
performs all of the updates on the grid. calls movement and controlls cell prop
*/
void updateMap(vector<Cell*>& grid) {
    for (int i = 0; i < (ROW * COL); i++) {
        grid[i]->setMoved(false); // it has not moved
    }

    for (int i = 0; i < (ROW * COL); i++) {
        int loc = grid[i]->move(grid, i);

        if ((loc != -1) && (grid[i]->getMoved() == false)) { // it has a move and it can move still

            grid[i]->setMoved(true); // It has now moved

            grid[loc] = grid[i]; //update location
            Cell* cell = new Cell();
            grid[i] = cell;
        }
        else {
            continue;
        }
    }

    // check for nazghul death
    for (int i = 0; i < (ROW * COL); i++) {
        if (grid[i]->getDead() == true) { // naz starved, replace with open cell
            Cell* cell = new Cell();
            grid[i] = cell;
        }
    }

    // for loop for reproduction goes here
    for (int i = 0; i < (ROW * COL); i++) {

        // if breed
        if (grid[i]->getBreed() == true) {
            grid[i]->setBreed(false); // I have tried to breed, I can no longer
            int loc = grid[i]->breeding(grid, i); // find a location to breed too
            // use this location to make a new hobbit
            if (loc != -1) { // use && to also check if the object is a hobbit
                if (grid[i]->getName() == 'O') {
                    Hobbit* hobbit = new Hobbit;
                    grid[loc] = hobbit;
                }
                // use the same logic as above for the nazghuls
                if (grid[i]->getName() == 'X') {
                    Nazghul* nazghul = new Nazghul;
                    grid[loc] = nazghul;
                }
                else {
                    continue;
                }
            }

        }
    }

}

/*
simply displays the grid
*/
void displayMap(vector<Cell*> grid) {

    for (int i = 0; i < (ROW * COL); i++) {
        if ((i % COL) == 0) { cout << "\n"; }
        if (grid[i]->getName() == 'X') { // adding the color
            changeColor(12);
        }
        else if ((grid[i]->getName() == 'O')) { // adding color
            changeColor(2);
        }
        else {
            changeColor(7); // adding color
        }
        cout << grid[i]->getName() << " ";
    }
};

/*
sets up inital conditions of population and includes update cycle.
*/
void runSim(vector<Cell*> grid) {
    // initate map
    initalConditions(grid);

    // inital display cycle
    displayMap(grid);

    // loop forever
    while (true) {
        Sleep(1000);;
        system("cls");
        updateMap(grid);
        displayMap(grid);
    }
```

```cpp
}

int main()
{
    srand(time(0));
    // make the grid
    vector<Cell*> grid = makeMap();

    // initalize & run the simulation
    runSim(grid);



}
```