```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
```

```
48              int le;
49              if ((loc % COL) == 0) { le = loc + (COL - 1); }
50              else { le = loc - 1; }
51              moves.push_back(le);
52
53              // right: +1 unless right is on the right side
54              int ri;
55              if ((loc % COL) == (COL - 1)) { ri = loc - (COL - 1); }
56              else { ri = loc + 1; }
57              moves.push_back(ri);
58
59              random_shuffle(moves.begin(), moves.end());
60              return moves;
61          };
62
63          /*
64          narrows down a list of posMoves (or a catered list) to spots on
              the grid that match
65          the char type returns the move or -1
66          */
67          int singleMove(vector<int> moves, vector<Cell*> grid, char type) {
68              for (int i = 0; i < moves.size(); i++) {
69                  if (grid[moves[i]]->getName() == type) {
70                      return moves[i];
71                  }
72              }
73              return -1;
74          }
75
76          /*
77          increases the age of the living, or resets age to 0 and seets
              breed to true
78          */
79          void ageUp(int max) {
80              if (age != max) {
81                  age ++;
82              }
83              else {
84                  breed = true;
85                  age = 0;
86              }
87          }
88
89      public:
90
91          Cell() { name = '.'; }
92
93          char getName() {
94              return name;
```

```
 95          }
 96          bool getMoved() {
 97              return moved;
 98          }
 99          bool getBreed() {
100              return breed;
101          }
102          bool getDead() {
103              return dead;
104          }
105          int getAge() {
106              return age;
107          }
108
109          void setBreed(bool bre) {
110              breed = bre;
111          }
112          void setMoved(bool mov) {
113              moved = mov;
114          }
115
116
117          virtual int move(vector<Cell*> grid, int loc) {
118              return -1;
119          }
120
121          virtual int breeding(vector<Cell*> grid, int loc) {
122              return -1;
123          }
124
125  };
126
127  class Hobbit : public Cell {
128      public:
129          Hobbit() : Cell() {
130              name = 'O';
131          };
132
133          int move(vector<Cell*> grid, int loc) {
134
135              // check to see if the age is 3. (the extra +1 due to calling
                    x2)
136              ageUp(3);
137
138              vector<int> moves = posMoves(loc); // the possable moves
139              return singleMove(moves, grid, '.');
140          }
141
142          int breeding(vector<Cell*> grid, int loc) {
```

```
143             vector<int> open = posMoves(loc); // the possable moves
144             return singleMove(open, grid, '.');
145         }
146 };
147
148 class Nazghul : public Cell {
149     private:
150         int lastFeed = 0; // how long from the last time it was fed
151
152
153         /*
154         clears the teleport moves off the move list, as the nazghuls can
              not teleport
155         */
156         vector<int> cleanMove(vector<int> rawMoves, int loc) {
157             vector<int> moves;
158
159             for (int i = 0; i < rawMoves.size(); i++) {
160                 if ((rawMoves[i] == (loc - 1)) || (rawMoves[i] == (loc -
                      COL)) ||
161                     (rawMoves[i] == (loc + 1)) || (rawMoves[i] == (loc +
                     COL))) {
162                     moves.push_back(rawMoves[i]);
163                 }
164             }
165             return moves;
166         }
167
168     public:
169         Nazghul() : Cell() {
170             name = 'X';
171         };
172
173
174         int move(vector<Cell*> grid, int loc) {
175
176             // track moved
177             // track age like with hobbit 0 - 8 inclusive
178             ageUp(8);
179
180             vector<int> moves = posMoves(loc); // for cleaning up options
                  not okay for nazzys
181             moves = cleanMove(moves, loc); // where the final moves will
                  go.
182             int move; // to hold the testable move
183
184             move = singleMove(moves, grid, 'O');
185             if (move != -1) {
186                 lastFeed = 0;
```

```
187                 return move;
188             }
189 
190             lastFeed++;
191             if (lastFeed == 3) { // per spec, above move
192                 dead = true;
193             }
194 
195             // while already determined dead, this is to allow it to
196             // emulate one last desperate search for food.
197             return singleMove(moves, grid, '.');
198         }
199 
200         int breeding(vector<Cell*> grid, int loc) {
201             vector<int> open = posMoves(loc); // the locs
202             open = cleanMove(open, loc);
203             int spot;
204 
205             spot = singleMove(open, grid, 'O');
206             if (spot != -1) {
207                 return spot;
208             }
209             return singleMove(open, grid, '.');
210         }
211 };
212 
213 /*
214 handler f(x)n for color change in console
215 */
216 void changeColor(int color) {
217     HANDLE hConsole;
218 
219     hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
220     SetConsoleTextAttribute(hConsole, color);
221 }
222 
223 /*
224 a 1d vector map. Map was made as a 1d vector to simplfiy project
225 */
226 vector<Cell*> makeMap() {
227     vector<Cell*> grid;
228 
229     for (int i = 0; i < ROW; i++) {
230         for (int j = 0; j < COL; j++) {
231 
232             Cell *cell = new Cell();
233             grid.push_back(cell);
234         }
235     }
```

```
236
237     return grid;
238 }
239
240 /*
241 seeds the map
242 */
243 vector<Cell*> initalConditions(vector<Cell*>& grid) {
244     vector<int> unique;
245     int counter = 0;
246
247     // make a list of unique indexs
248     while (counter < (HOB + NAZ)) {
249
250         int testIndex = rand() % (ROW * COL); // random number in range of grid
251         bool used = false; // is the index already taken?
252
253         for (int i = 0; i < counter; i++) {
254             if (unique[i] == testIndex) { used = true; } // if used is true, the number is garbage
255         }
256         if (used == false) {
257             unique.push_back(testIndex);
258             counter ++;
259         }
260     }
261
262     // use constants from above
263     for (int i = 0; i < unique.size(); i++) {
264         int loc = unique[i];
265         if (i < NAZ) {
266             Nazghul *naz = new Nazghul();
267             grid[loc] = naz;
268         }
269         else {
270             Hobbit *hob = new Hobbit();
271             grid[loc] = hob;
272         }
273     }
274
275     return grid;
276 }
277
278 /*
279 performs all of the updates on the grid. calls movement and controlls cell prop
280 */
281 void updateMap(vector<Cell*>& grid) {
```

```
282     for (int i = 0; i < (ROW * COL); i++) {
283         grid[i]->setMoved(false); // it has not moved
284     }
285
286     for (int i = 0; i < (ROW * COL); i++) {
287         int loc = grid[i]->move(grid, i);
288
289         if ((loc != -1) && (grid[i]->getMoved() == false)) { // it has a
              move and it can move still
290
291             grid[i]->setMoved(true); // It has now moved
292
293             grid[loc] = grid[i]; //update location
294             Cell* cell = new Cell();
295             grid[i] = cell;
296         }
297         else {
298             continue;
299         }
300     }
301
302     // check for nazghul death
303     for (int i = 0; i < (ROW * COL); i++) {
304         if (grid[i]->getDead() == true) { // naz starved, replace with
              open cell
305             Cell* cell = new Cell();
306             grid[i] = cell;
307         }
308     }
309
310     // for loop for reproduction goes here
311     for (int i = 0; i < (ROW * COL); i++) {
312
313         // if breed
314         if (grid[i]->getBreed() == true) {
315             grid[i]->setBreed(false); // I have tried to breed, I can no
                  longer
316             int loc = grid[i]->breeding(grid, i); // find a location to
                  breed too
317             // use this location to make a new hobbit
318             if (loc != -1) { // use && to also check if the object is a
                  hobbit
319                 if (grid[i]->getName() == 'O') {
320                     Hobbit* hobbit = new Hobbit;
321                     grid[loc] = hobbit;
322                 }
323                 // use the same logic as above for the nazghuls
324                 if (grid[i]->getName() == 'X') {
325                     Nazghul* nazghul = new Nazghul;
```

```
326                     grid[loc] = nazghul;
327                 }
328                 else {
329                     continue;
330                 }
331             }
332 
333         }
334     }
335 
336 }
337 
338 /*
339 simply displays the grid
340 */
341 void displayMap(vector<Cell*> grid) {
342 
343     for (int i = 0; i < (ROW * COL); i++) {
344         if ((i % COL) == 0) { cout << "\n"; }
345         if (grid[i]->getName() == 'X') { // adding the color
346             changeColor(12);
347         }
348         else if ((grid[i]->getName() == 'O')) { // adding color
349             changeColor(2);
350         }
351         else {
352             changeColor(7); // adding color
353         }
354         cout << grid[i]->getName() << " ";
355     }
356 };
357 
358 /*
359 sets up inital conditions of population and includes update cycle.
360 */
361 void runSim(vector<Cell*> grid) {
362     // initate map
363     initalConditions(grid);
364 
365     // inital display cycle
366     displayMap(grid);
367 
368     // loop forever
369     while (true) {
370         Sleep(1000);;
371         system("cls");
372         updateMap(grid);
373         displayMap(grid);
374     }
```

```
375 }
376
377 int main()
378 {
379     srand(time(0));
380     // make the grid
381     vector<Cell*> grid = makeMap();
382
383     // initalize & run the simulation
384     runSim(grid);
385
386
387
388 }
389
390
```