#include <iostream>

#include <string>

#include <random>

#include <utility>

#include <cstdlib>

using namespace std;

///////////////////////////////////

// Manifest constants

const int MAXROWS = 20; // max number of rows in the arena

const int MAXCOLS = 30; // max number of columns in the arena

const int MAXZOMBIES = 150; // max number of zombies allowed

const int UP = 0;

const int DOWN = 1;

const int LEFT = 2;

const int RIGHT = 3;

const int NUMDIRS = 4;

///////////////////////////////////

// Auxiliary function declarations

int decodeDirection(char dir);

int randInt(int min, int max);

void clearScreen();

///////////////////////////////////

// Type definitions

class Arena; // let the compiler know Arena is a type name, since it's mentioned in the Zombie declaration.

class Zombie

{

public:

// Constructor

Zombie(Arena\* ap, int r, int c);

// Accessors

int row() const;

int col() const;

// Mutators

void move();

bool getAttacked(int dir);

private:

Arena\* m\_arena;

int m\_row;

int m\_col;

int m\_attacks;

};

class Player

{

public:

// Constructor

Player(Arena \*ap, int r, int c);

// Accessors

int row() const;

int col() const;

int age() const;

bool isDead() const;

// Mutators

void stand();

void moveOrAttack(int dir);

void setDead();

private:

Arena\* m\_arena;

int m\_row;

int m\_col;

int m\_age;

bool m\_dead;

};

class Arena

{

public:

// Constructor/destructor

Arena(int nRows, int nCols);

~Arena();

// Accessors

int rows() const;

int cols() const;

Player\* player() const;

int zombieCount() const;

int numZombiesAt(int r, int c) const;

bool determineNewPosition(int& r, int& c, int dir) const;

void display() const;

// Mutators

bool addZombie(int r, int c);

bool addPlayer(int r, int c);

bool attackZombieAt(int r, int c, int dir);

bool moveZombies();

private:

int m\_rows;

int m\_cols;

Player\* m\_player;

Zombie\* m\_zombies[MAXZOMBIES];

int m\_nZombies;

};

class Game

{

public:

// Constructor/destructor

Game(int rows, int cols, int nZombies);

~Game();

// Mutators

void play();

private:

Arena\* m\_arena;

};

///////////////////////////////////

// Zombie implementation

Zombie::Zombie(Arena\* ap, int r, int c)

{

if (ap == nullptr)

{

cout << "\*\*\*\*\* A zombie must be created in some Arena!" << endl;

exit(1);

}

if (r < 1 || r > ap->rows() || c < 1 || c > ap->cols())

{

cout << "\*\*\*\*\* Zombie created with invalid coordinates (" << r << ","

<< c << ")!" << endl;

exit(1);

}

m\_arena = ap;

m\_row = r;

m\_col = c;

m\_attacks = 0;

}

int Zombie::row() const

{

return m\_row;

}

int Zombie::col() const

{

return m\_col;

}

void Zombie::move()

{

// Attempt to move in a random direction; if we can't move, don't move

int dir = randInt(0, NUMDIRS - 1); // dir is now UP, DOWN, LEFT, or RIGHT

switch (dir)

{

case UP:

if (m\_row - 1 == 0)

break;

m\_row--;

break;

case DOWN:

if (m\_row + 1 > m\_arena->rows())

break;

m\_row++;

break;

case LEFT:

if (m\_col - 1 == 0)

break;

m\_col--;

break;

case RIGHT:

if (m\_col + 1 > m\_arena->cols())

break;

m\_col++;

break;

}

// Attempt to move in direction dir; if we can't move, don't move.

}

bool Zombie::getAttacked(int dir) // return true if dies

{

if (m\_attacks == 1)

return true;

if (m\_arena->determineNewPosition(m\_row, m\_col, dir))

{

m\_attacks++;

return false;

}

// If the zombie has been attacked once before, return true

// (since a second attack destroys a zombie). Otherwise, if possible,

// move the zombie in one position in direction dir and return false

// (since it survived the injury). Otherwise, do not move, but return

// true (since the momentum from the blow would bump the zombie against

// the wall, dealing it an additional fatal injury).

return true;

}

///////////////////////////////////

// Player implementations

Player::Player(Arena\* ap, int r, int c)

{

if (ap == nullptr)

{

cout << "\*\*\*\*\* The player must be created in some Arena!" << endl;

exit(1);

}

if (r < 1 || r > ap->rows() || c < 1 || c > ap->cols())

{

cout << "\*\*\*\* Player created with invalid coordinates (" << r

<< "," << c << ")!" << endl;

exit(1);

}

m\_arena = ap;

m\_row = r;

m\_col = c;

m\_age = 0;

m\_dead = false;

}

int Player::row() const

{

return m\_row;

}

int Player::col() const

{

return m\_col;

}

int Player::age() const

{

return m\_age;

}

void Player::stand()

{

m\_age++;

}

void Player::moveOrAttack(int dir)

{

m\_age++;

if (m\_arena->numZombiesAt(m\_row, m\_col) > 0)

m\_arena->attackZombieAt(m\_row, m\_col, dir);

else

switch (dir)

{

case UP:

if (m\_arena->numZombiesAt(m\_row - 1, m\_col) > 0)

m\_arena->attackZombieAt(m\_row - 1, m\_col, dir);

else if (m\_row - 1 == 0)

break;

else

m\_row--;

break;

case DOWN:

if (m\_arena->numZombiesAt(m\_row + 1, m\_col) > 0)

m\_arena->attackZombieAt(m\_row + 1, m\_col, dir);

else if (m\_row + 1 > m\_arena->rows())

break;

else

m\_row++;

break;

case LEFT:

if (m\_arena->numZombiesAt(m\_row, m\_col - 1) > 0)

m\_arena->attackZombieAt(m\_row, m\_col - 1, dir);

else if (m\_col - 1 == 0)

break;

else m\_col--;

break;

case RIGHT:

if (m\_arena->numZombiesAt(m\_row, m\_col + 1) > 0)

m\_arena->attackZombieAt(m\_row, m\_col + 1, dir);

else if (m\_col + 1 > m\_arena->cols())

break;

else

m\_col++;

break;

}

// If there is a zombie adjacent to the player in the direction

// dir, attack it. Otherwise, move the player to that position if

// possible (i.e., if the move would not be off the edge of the arena).

}

bool Player::isDead() const

{

return m\_dead;

}

void Player::setDead()

{

m\_dead = true;

}

///////////////////////////////////

// Arena implementations

Arena::Arena(int nRows, int nCols)

{

if (nRows <= 0 || nCols <= 0 || nRows > MAXROWS || nCols > MAXCOLS)

{

cout << "\*\*\*\*\* Arena created with invalid size " << nRows << " by "

<< nCols << "!" << endl;

exit(1);

}

m\_rows = nRows;

m\_cols = nCols;

m\_player = nullptr;

m\_nZombies = 0;

}

Arena::~Arena()

{

// Delete the player and all remaining dynamically allocated zombies.

delete m\_player;

while (m\_nZombies != 0)

{

delete (m\_zombies[m\_nZombies - 1]);

m\_nZombies--;

}

}

int Arena::rows() const

{

return m\_rows;

}

int Arena::cols() const

{

return m\_cols;

}

Player\* Arena::player() const

{

return m\_player;

}

int Arena::zombieCount() const

{

return m\_nZombies;

}

int Arena::numZombiesAt(int r, int c) const

{

int i = 0, count = 0;

while (i != m\_nZombies)

{

if (c == m\_zombies[i]->col())

if (r == m\_zombies[i]->row())

count++;

i++;

}// Return the number of zombies at row r, column c.

return count;

}

bool Arena::determineNewPosition(int& r, int& c, int dir) const

{

switch (dir)

{

case UP:

if (r == 1)

return false;

else

r--;

break;

case DOWN:

if (r == m\_rows)

return false;

else

r++;

break;

case LEFT:

if (c == 1)

return false;

else

c--;

break;

case RIGHT:

if (c == m\_cols)

return false;

else

c++;

break;

default:

return false;

}

return true;

// If a move from row r, column c, one step in direction dir

// would go off the edge of the arena, leave r and c unchanged and

// return false. Otherwise, set r or c so that row r, column c, is

// now the new position resulting from the proposed move, and

// return true.

}

void Arena::display() const

{

// Position (row,col) in the arena coordinate system is represented in

// the array element grid[row-1][col-1]

char grid[MAXROWS][MAXCOLS];

int r, c;

// Fill the grid with dots

for (r = 0; r < rows(); r++)

for (c = 0; c < cols(); c++)

grid[r][c] = '.';

// Indicate each zombie's position

// If one zombie is at some grid point, set the char to 'Z'.

// If it's 2 though 8, set it to '2' through '8'.

// For 9 or more, set it to '9'.

for (r = 0; r < rows(); r++)

{

for (c = 0; c < cols(); c++)

{

if (numZombiesAt(r + 1, c + 1) == 1)

grid[r][c] = 'Z';

else if (numZombiesAt(r + 1, c + 1) <= 8 && numZombiesAt(r + 1, c + 1) >= 2)

grid[r][c] = '0' + numZombiesAt(r + 1, c + 1);

else if (numZombiesAt(r + 1, c + 1) >= 9)

grid[r][c] = '0' + numZombiesAt(r + 1, c + 1);

}

}

// Indicate player's position

if (m\_player != nullptr)

{

// Set the char to '@', unless there's also a zombie there,

// in which case set it to '\*'.

char& gridChar = grid[m\_player->row() - 1][m\_player->col() - 1];

if (gridChar == '.')

gridChar = '@';

else

gridChar = '\*';

}

// Draw the grid

clearScreen();

for (r = 0; r < rows(); r++)

{

for (c = 0; c < cols(); c++)

cout << grid[r][c];

cout << endl;

}

cout << endl;

// Write message, zombie, and player info

cout << endl;

cout << "There are " << zombieCount() << " zombies remaining." << endl;

if (m\_player == nullptr)

cout << "There is no player." << endl;

else

{

if (m\_player->age() > 0)

cout << "The player has lasted " << m\_player->age() << " steps." << endl;

if (m\_player->isDead())

cout << "The player is dead." << endl;

}

}

bool Arena::addZombie(int r, int c)

{

// If there are MAXZOMBIES zombies, return false. Otherwise,

// dynamically allocate a new Zombie at coordinates (r,c). Save the

// pointer to the newly allocated Zombie and return true.

// Implement this.

if (m\_nZombies == MAXZOMBIES)

return false;

m\_zombies[m\_nZombies] = new Zombie(this, r, c);

m\_nZombies++;

return true;

}

bool Arena::addPlayer(int r, int c)

{

// Don't add a player if one already exists

if (m\_player != nullptr)

return false;

// Dynamically allocate a new Player and add it to the arena

m\_player = new Player(this, r, c);

return true;

}

bool Arena::attackZombieAt(int r, int c, int dir)

{

int i;

if (numZombiesAt(r, c) == 0)

return false;

for (i = 0; i < m\_nZombies; i++)

{

if (m\_zombies[i]->row() == r)

{

if (m\_zombies[i]->col() == c)

{

if (m\_zombies[i]->getAttacked(dir))

{

delete m\_zombies[i];

m\_zombies[i] = m\_zombies[m\_nZombies - 1];

m\_nZombies--;

return true;

}

break;

}

}

}

// Attack one zombie at row r, column c if at least one is at

// that position. If the zombie does not survive the injury, destroy the

// zombie object, removing it from the arena, and return true. Otherwise,

// return false (no zombie at (r,c), or zombie didn't die).

return false;

}

bool Arena::moveZombies()

{

for (int k = 0; k < m\_nZombies; k++)

{

m\_zombies[k]->move();

if (m\_zombies[k]->row() == m\_player->row())

if (m\_zombies[k]->col() == m\_player->col())

m\_player->setDead();

// Have the k-th zombie in the arena make one move.

// If that move results in that zombie being in the same

// position as the player, the player dies.

}

return !m\_player->isDead();

}

///////////////////////////////////

// Game implementations

Game::Game(int rows, int cols, int nZombies)

{

if (nZombies < 0)

{

cout << "\*\*\*\*\* Cannot create Game with negative number of zombies!" << endl;

exit(1);

}

if (nZombies > MAXZOMBIES)

{

cout << "\*\*\*\*\* Trying to create Game with " << nZombies

<< " zombies; only " << MAXZOMBIES << " are allowed!" << endl;

exit(1);

}

if (rows == 1 && cols == 1 && nZombies > 0)

{

cout << "\*\*\*\*\* Cannot create Game with nowhere to place the zombies!" << endl;

exit(1);

}

// Create arena

m\_arena = new Arena(rows, cols);

// Add player

int rPlayer = randInt(1, rows);

int cPlayer = randInt(1, cols);

m\_arena->addPlayer(rPlayer, cPlayer);

// Populate with zombies

while (nZombies > 0)

{

int r = randInt(1, rows);

int c = randInt(1, cols);

// Don't put a zombie where the player is

if (r == rPlayer && c == cPlayer)

continue;

m\_arena->addZombie(r, c);

nZombies--;

}

}

Game::~Game()

{

delete m\_arena;

}

void Game::play()

{

m\_arena->display();

Player\* p = m\_arena->player();

if (p == nullptr)

return;

while (!m\_arena->player()->isDead() && m\_arena->zombieCount() > 0)

{

cout << endl;

cout << "Move (u/d/l/r//q): ";

string action;

getline(cin, action);

if (action.size() == 0) // player stands

p->stand();

else

{

switch (action[0])

{

default: // if bad move, nobody moves

cout << '\a' << endl; // beep

continue;

case 'q':

return;

case 'u':

case 'd':

case 'l':

case 'r':

p->moveOrAttack(decodeDirection(action[0]));

break;

}

}

m\_arena->moveZombies();

m\_arena->display();

}

}

///////////////////////////////////

// Auxiliary function implementations

int decodeDirection(char dir)

{

switch (dir)

{

case 'u': return UP;

case 'd': return DOWN;

case 'l': return LEFT;

case 'r': return RIGHT;

}

return -1; // bad argument passed in!

}

// Return a random int from min to max, inclusive

int randInt(int min, int max)

{

if (max < min)

swap(max, min);

static random\_device rd;

static default\_random\_engine generator(rd());

uniform\_int\_distribution<> distro(min, max);

return distro(generator);

}

///////////////////////////////////

// main()

int main()

{

Game g(7, 8, 25);

g.play();

}