```
//
//  main.cpp
//  Project 7
//
//  Created by Christopher Clark on 12/4/19.
//  Copyright © 2019 Christopher Clark. All rights reserved.
//

// vampires.cpp

// Portions you are to complete are marked with a TODO: comment.
// We've provided some incorrect return statements (so indicated) just
// to allow this skeleton program to compile and run, albeit incorrectly.
// The first thing you probably want to do is implement the utterly trivial
// functions (marked TRIVIAL).  Then get Arena::display going.  That gives
// you more flexibility in the order you tackle the rest of the functionality.
// As you finish implementing each TODO: item, remove its TODO: comment.

#include <iostream>
#include <string>
#include <random>
#include <utility>
#include <cstdlib>
#include <cctype>
using namespace std;

///////////////////////////////////////////////////////////////////////
// Manifest constants
///////////////////////////////////////////////////////////////////////

const int MAXROWS = 20;               // max number of rows in the arena
const int MAXCOLS = 20;               // max number of columns in the arena
const int MAXVAMPIRES = 100;          // max number of vampires allowed

const int NORTH = 0;
const int EAST  = 1;
const int SOUTH = 2;
const int WEST  = 3;
const int NUMDIRS = 4;

const int EMPTY      = 0;
const int HAS_POISON = 1;

///////////////////////////////////////////////////////////////////////
// Type definitions
///////////////////////////////////////////////////////////////////////

class Arena;  // This is needed to let the compiler know that Arena is a
              // type name, since it's mentioned in the Vampire declaration.

class Vampire
```

```
{
  public:
      // Constructor
    Vampire(Arena* ap, int r, int c);

      // Accessors
    int  row() const;
    int  col() const;
    bool isDead() const;

      // Mutators
    void move();

  private:
    Arena* m_arena;
    int    m_row;
    int    m_col;
      // TODO: You'll probably find that a vampire object needs additional
      // data members to support your implementation of the behavior affected
      // by poisoned blood vials.
};

class Player
{
  public:
      // Constructor
    Player(Arena* ap, int r, int c);

      // Accessors
    int  row() const;
    int  col() const;
    bool isDead() const;

      // Mutators
    string dropPoisonVial();
    string move(int dir);
    void   setDead();

  private:
    Arena* m_arena;
    int    m_row;
    int    m_col;
    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     vampireCount() const;
    int     getCellStatus(int r, int c) const;
    int     numberOfVampiresAt(int r, int c) const;
    void    display(string msg) const;

      // Mutators
    void setCellStatus(int r, int c, int status);
    bool addVampire(int r, int c);
    bool addPlayer(int r, int c);
    void moveVampires();

  private:
    int      m_grid[MAXROWS][MAXCOLS];
    int      m_rows;
    int      m_cols;
    Player*  m_player;
    Vampire* m_vampires[MAXVAMPIRES];
    int      m_nVampires;
    int      m_turns;

      // Helper functions
    void checkPos(int r, int c, string functionName) const;
    bool isPosInBounds(int r, int c) const;
};

class Game
{
  public:
      // Constructor/destructor
    Game(int rows, int cols, int nVampires);
    ~Game();

      // Mutators
    void play();

  private:
    Arena* m_arena;

      // Helper functions
    string takePlayerTurn();
};

///////////////////////////////////////////////////////////////////////
//  Auxiliary function declarations
///////////////////////////////////////////////////////////////////////
```

```
int randInt(int lowest, int highest);
bool decodeDirection(char ch, int& dir);
bool attemptMove(const Arena& a, int dir, int& r, int& c);
bool recommendMove(const Arena& a, int r, int c, int& bestDir);
void clearScreen();

///////////////////////////////////////////////////////////////////////
//  Vampire implementation
///////////////////////////////////////////////////////////////////////

Vampire::Vampire(Arena* ap, int r, int c)
{
    if (ap == nullptr)
    {
        cout << "***** A vampire must be created in some Arena!" << endl;
        exit(1);
    }
    if (r < 1  ||  r > ap->rows()  ||  c < 1  ||  c > ap->cols())
    {
        cout << "***** Vampire created with invalid coordinates (" << r << ","
             << c << ")!" << endl;
        exit(1);
    }
    m_arena = ap;
    m_row = r;
    m_col = c;
}

int Vampire::row() const
{
    return m_row;
}

int Vampire::col() const
{
      // TODO: TRIVIAL:  Return what column the Vampire is at
      // Delete the following line and replace it with the correct code.
    return 1;  // This implementation compiles, but is incorrect.
}

bool Vampire::isDead() const
{
      // TODO: Return whether the Vampire is dead
      // Delete the following line and replace it with the correct code.
    return false;  // This implementation compiles, but is incorrect.
}

void Vampire::move()
{
      // TODO:
      //   Return without moving if the vampire has drunk one vial of
```

```
      //   poisoned blood (so is supposed to move only every other turn) and
      //   this is a turn it does not move.

      //   Otherwise, attempt to move in a random direction; if can't
      //   move, don't move.  If it lands on a poisoned blood vial, drink all
      //   the blood in the vial and remove it from the game (so it is no
      //   longer on that grid point).
}

///////////////////////////////////////////////////////////////////////
//  Player implementation
///////////////////////////////////////////////////////////////////////

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_dead = false;
}

int Player::row() const
{
      // TODO: TRIVIAL:  Return what row the Player is at
      // Delete the following line and replace it with the correct code.
    return 1;  // This implementation compiles, but is incorrect.
}

int Player::col() const
{
      // TODO: TRIVIAL:  Return what column the Player is at
      // Delete the following line and replace it with the correct code.
    return 1;  // This implementation compiles, but is incorrect.
}

string Player::dropPoisonVial()
{
    if (m_arena->getCellStatus(m_row, m_col) == HAS_POISON)
        return "There's already a poisoned blood vial at this spot.";
    m_arena->setCellStatus(m_row, m_col, HAS_POISON);
```

```
    return "A poisoned blood vial has been dropped.";
}

string Player::move(int dir)
{
      // TODO:  Attempt to move the player one step in the indicated
      //        direction.  If this fails,
      //        return "Player couldn't move; player stands."
      //        A player who moves onto a vampire dies, and this
      //        returns "Player walked into a vampire and died."
      //        Otherwise, return one of "Player moved north.",
      //        "Player moved east.", "Player moved south.", or
      //        "Player moved west."
    return "Player couldn't move; player stands.";  // This implementation
     compiles, but is incorrect.
}

bool Player::isDead() const
{
      // TODO: Return whether the Player is dead
      // Delete the following line and replace it with the correct code.
    return false;  // This implementation compiles, but is incorrect.
}

void Player::setDead()
{
    m_dead = true;
}

///////////////////////////////////////////////////////////////////////
//  Arena implementation
///////////////////////////////////////////////////////////////////////

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_nVampires = 0;
    m_turns = 0;
    for (int r = 1; r <= m_rows; r++)
        for (int c = 1; c <= m_cols; c++)
            setCellStatus(r, c, EMPTY);
}
```

```
Arena::~Arena()
{
    // TODO:  Deallocate the player and all remaining dynamically allocated
    //        vampires.
}

int Arena::rows() const
{
      // TODO: TRIVIAL:  Return the number of rows in the arena
      // Delete the following line and replace it with the correct code.
    return 1;  // This implementation compiles, but is incorrect.
}

int Arena::cols() const
{
      // TODO: TRIVIAL:  Return the number of columns in the arena
      // Delete the following line and replace it with the correct code.
    return 1;  // This implementation compiles, but is incorrect.
}

Player* Arena::player() const
{
    return m_player;
}

int Arena::vampireCount() const
{
      // TODO: TRIVIAL:  Return the number of vampires in the arena
      // Delete the following line and replace it with the correct code.
    return 0;  // This implementation compiles, but is incorrect.
}

int Arena::getCellStatus(int r, int c) const
{
    checkPos(r, c, "Arena::getCellStatus");
    return m_grid[r-1][c-1];
}

int Arena::numberOfVampiresAt(int r, int c) const
{
      // TODO:  Return the number of vampires at row r, column c
      // Delete the following line and replace it with the correct code.
    return 0;  // This implementation compiles, but is incorrect.
}

void Arena::display(string msg) const
{
    char displayGrid[MAXROWS][MAXCOLS];
    int r, c;

      // Fill displayGrid with dots (empty) and stars (poisoned blood vials)
```

```
    for (r = 1; r <= rows(); r++)
        for (c = 1; c <= cols(); c++)
            displayGrid[r-1][c-1] = (getCellStatus(r,c) == EMPTY ? '.' : '*');

      // Indicate each vampire's position
      // TODO:  If one vampire is at some grid point, set the displayGrid char
      //        to 'V'.  If it's 2 though 8, set it to '2' through '8'.
      //        For 9 or more, set it to '9'.

      // Indicate player's position
    if (m_player != nullptr)
        displayGrid[m_player->row()-1][m_player->col()-1] =
         (m_player->isDead() ? 'X' : '@');

      // Draw the grid
    clearScreen();
    for (r = 1; r <= rows(); r++)
    {
        for (c = 1; c <= cols(); c++)
            cout << displayGrid[r-1][c-1];
        cout << endl;
    }
    cout << endl;

      // Write message, vampire, and player info
    if (msg != "")
        cout << msg << endl;
    cout << "There are " << vampireCount() << " vampires remaining." << endl;
    if (m_player == nullptr)
        cout << "There is no player!" << endl;
    else if (m_player->isDead())
        cout << "The player is dead." << endl;
    cout << m_turns << " turns have been taken." << endl;
}

void Arena::setCellStatus(int r, int c, int status)
{
    checkPos(r, c, "Arena::setCellStatus");
    m_grid[r-1][c-1] = status;
}

bool Arena::addVampire(int r, int c)
{
    if (! isPosInBounds(r, c))
        return false;

      // Don't add a vampire on a spot with a poisoned blood vial
    if (getCellStatus(r, c) != EMPTY)
        return false;

      // Don't add a vampire on a spot with a player
```

```
    if (m_player != nullptr  &&  m_player->row() == r  &&  m_player->col() ==
     c)
        return false;

      // If there are MAXVAMPIRES existing vampires, return false.  Otherwise,
      // dynamically allocate a new vampire at coordinates (r,c).  Save the
      // pointer to newly allocated vampire and return true.

      // TODO:  Implement this.
    return false;  // This implementation compiles, but is incorrect.
}

bool Arena::addPlayer(int r, int c)
{
    if (! isPosInBounds(r, c))
        return false;

      // Don't add a player if one already exists
    if (m_player != nullptr)
        return false;

      // Don't add a player on a spot with a vampire
    if (numberOfVampiresAt(r, c) > 0)
        return false;

    m_player = new Player(this, r, c);
    return true;
}

void Arena::moveVampires()
{
      // Move all vampires
      // TODO:  Move each vampire.  Mark the player as dead if necessary.
      //        Deallocate any dead dynamically allocated vampire.

      // Another turn has been taken
    m_turns++;
}

bool Arena::isPosInBounds(int r, int c) const
{
    return (r >= 1  &&  r <= m_rows  &&  c >= 1  &&  c <= m_cols);
}

void Arena::checkPos(int r, int c, string functionName) const
{
    if (r < 1  ||  r > m_rows  ||  c < 1  ||  c > m_cols)
    {
        cout << "***** " << "Invalid arena position (" << r << ","
             << c << ") in call to " << functionName << endl;
        exit(1);
```

```
    }
}

///////////////////////////////////////////////////////////////////////
//  Game implementation
///////////////////////////////////////////////////////////////////////

Game::Game(int rows, int cols, int nVampires)
{
    if (nVampires < 0)
    {
        cout << "***** Cannot create Game with negative number of vampires!"
         << endl;
        exit(1);
    }
    if (nVampires > MAXVAMPIRES)
    {
        cout << "***** Trying to create Game with " << nVampires
             << " vampires; only " << MAXVAMPIRES << " are allowed!" << endl;
        exit(1);
    }
    int nEmpty = rows * cols - nVampires - 1;  // 1 for Player
    if (nEmpty < 0)
    {
        cout << "***** Game created with a " << rows << " by "
             << cols << " arena, which is too small too hold a player and "
             << nVampires << " vampires!" << endl;
        exit(1);
    }

      // Create arena
    m_arena = new Arena(rows, cols);

      // Add player
    int rPlayer;
    int cPlayer;
    do
    {
        rPlayer = randInt(1, rows);
        cPlayer = randInt(1, cols);
    } while (m_arena->getCellStatus(rPlayer, cPlayer) != EMPTY);
    m_arena->addPlayer(rPlayer, cPlayer);

      // Populate with vampires
    while (nVampires > 0)
    {
        int r = randInt(1, rows);
        int c = randInt(1, cols);
        if (r == rPlayer && c == cPlayer)
            continue;
        m_arena->addVampire(r, c);
```

```
        nVampires--;
    }
}

Game::~Game()
{
    delete m_arena;
}

string Game::takePlayerTurn()
{
    for (;;)
    {
        cout << "Your move (n/e/s/w/x or nothing): ";
        string playerMove;
        getline(cin, playerMove);

        Player* player = m_arena->player();
        int dir;

        if (playerMove.size() == 0)
        {
            if (recommendMove(*m_arena, player->row(), player->col(), dir))
                return player->move(dir);
            else
                return player->dropPoisonVial();
        }
        else if (playerMove.size() == 1)
        {
            if (tolower(playerMove[0]) == 'x')
                return player->dropPoisonVial();
            else if (decodeDirection(playerMove[0], dir))
                return player->move(dir);
        }
        cout << "Player move must be nothing, or 1 character n/e/s/w/x." <<
         endl;
    }
}

void Game::play()
{
    m_arena->display("");
    Player* player = m_arena->player();
    if (player == nullptr)
        return;
    while ( ! player->isDead()  &&  m_arena->vampireCount() > 0)
    {
        string msg = takePlayerTurn();
        m_arena->display(msg);
        if (player->isDead())
            break;
```

```
        m_arena->moveVampires();
        m_arena->display(msg);
    }
    if (player->isDead())
        cout << "You lose." << endl;
    else
        cout << "You win." << endl;
}

///////////////////////////////////////////////////////////
//  Auxiliary function implementation
///////////////////////////////////////////////////////////

  // Return a uniformly distributed random int from lowest to highest,
   inclusive
int randInt(int lowest, int highest)
{
    if (highest < lowest)
        swap(highest, lowest);
    static random_device rd;
    static default_random_engine generator(rd());
    uniform_int_distribution<> distro(lowest, highest);
    return distro(generator);
}

bool decodeDirection(char ch, int& dir)
{
    switch (tolower(ch))
    {
      default:  return false;
      case 'n': dir = NORTH; break;
      case 'e': dir = EAST;  break;
      case 's': dir = SOUTH; break;
      case 'w': dir = WEST;  break;
    }
    return true;
}

  // Return false without changing anything if moving one step from (r,c)
  // in the indicated direction would run off the edge of the arena.
  // Otherwise, update r and c to the position resulting from the move and
  // return true.
bool attemptMove(const Arena& a, int dir, int& r, int& c)
{
      // TODO:  Implement this function
      // Delete the following line and replace it with the correct code.
    return false;  // This implementation compiles, but is incorrect.
}

  // Recommend a move for a player at (r,c):  A false return means the
  // recommendation is that the player should drop a poisoned blood vial and
```

```
  // not move; otherwise, this function sets bestDir to the recommended
  // direction to move and returns true.
bool recommendMove(const Arena& a, int r, int c, int& bestDir)
{
      // TODO:  Implement this function
      // Delete the following line and replace it with your code.
    return false;  // This implementation compiles, but is incorrect.

      // Your replacement implementation should do something intelligent.
      // You don't have to be any smarter than the following, although
      // you can if you want to be:  If staying put runs the risk of a
      // vampire possibly moving onto the player's location when the vampires
      // move, yet moving in a particular direction puts the player in a
      // position that is safe when the vampires move, then the chosen
      // action is to move to a safer location.  Similarly, if staying put
      // is safe, but moving in certain directions puts the player in
      // danger of dying when the vampires move, then the chosen action should
      // not be to move in one of the dangerous directions; instead, the player
      // should stay put or move to another safe position.  In general, a
      // position that may be moved to by many vampires is more dangerous than
      // one that may be moved to by few.
      //
      // Unless you want to, you do not have to take into account that a
      // vampire might be poisoned and thus sometimes less dangerous than one
      // that is not.  That requires a more sophisticated analysis that
      // we're not asking you to do.
}

///////////////////////////////////////////////////////////////////////
// main()
///////////////////////////////////////////////////////////////////////

int main()
{
      // Create a game
      // Use this instead to create a mini-game:   Game g(3, 5, 2);
    Game g(10, 12, 40);

      // Play the game
    g.play();
}

///////////////////////////////////////////////////////////////////////
//  clearScreen implementation
///////////////////////////////////////////////////////////////////////

// DO NOT MODIFY OR REMOVE ANY CODE BETWEEN HERE AND THE END OF THE FILE!!!
// THE CODE IS SUITABLE FOR VISUAL C++, XCODE, AND g++/g31 UNDER LINUX.

// Note to Xcode users:  clearScreen() will just write a newline instead
// of clearing the window if you launch your program from within Xcode.
```

```
// That's acceptable.  (The Xcode output window doesn't have the capability
// of being cleared.)

#ifdef _MSC_VER  //  Microsoft Visual C++

#pragma warning(disable : 4005)
#include <windows.h>

void clearScreen()
{
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    CONSOLE_SCREEN_BUFFER_INFO csbi;
    GetConsoleScreenBufferInfo(hConsole, &csbi);
    DWORD dwConSize = csbi.dwSize.X * csbi.dwSize.Y;
    COORD upperLeft = { 0, 0 };
    DWORD dwCharsWritten;
    FillConsoleOutputCharacter(hConsole, TCHAR(' '), dwConSize, upperLeft,
                                                        &dwCharsWritten);
    SetConsoleCursorPosition(hConsole, upperLeft);
}

#else  // not Microsoft Visual C++, so assume UNIX interface

#include <iostream>
#include <cstring>
#include <cstdlib>

void clearScreen()  // will just write a newline in an Xcode output window
{
    static const char* term = getenv("TERM");
    if (term == nullptr  ||  strcmp(term, "dumb") == 0)
        cout << endl;
    else
    {
        static const char* ESC_SEQ = "\x1B[";  // ANSI Terminal esc seq:  ESC [
        cout << ESC_SEQ << "2J" << ESC_SEQ << "H" << flush;
    }
}

#endif
```