```
// vampires.cpp

#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 INITIAL_VAMPIRE_HEALTH = 2;  // initial vampire health
const int POISONED_IDLE_TIME = 1;       // poisoned vampire idles this many turns
                                        //   between moves

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;
    int    m_health;
    int    m_idleTurnsRemaining;
};

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);
int computeDanger(const Arena& a, int r, int c);
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;
    m_health = INITIAL_VAMPIRE_HEALTH;
    m_idleTurnsRemaining = 0;
}

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

int Vampire::col() const
{
    return m_col;
}

bool Vampire::isDead() const
{
    return m_health == 0;
}

void Vampire::move()
{
    if (m_idleTurnsRemaining > 0)
    {
        m_idleTurnsRemaining--;
        return;
    }

      // Attempt to move in a random direction; if we can't move, don't move
    if (attemptMove(*m_arena, randInt(0, NUMDIRS-1), m_row, m_col))
    {
        if (m_arena->getCellStatus(m_row, m_col) == HAS_POISON)
        {
            m_arena->setCellStatus(m_row, m_col, EMPTY);
            m_health--;
        }
    }

    if (m_health < INITIAL_VAMPIRE_HEALTH)
        m_idleTurnsRemaining = POISONED_IDLE_TIME;
}

///////////////////////////////////////////////////////////////////////////
//  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
{
    return m_row;
}

int Player::col() const
{
    return m_col;
}

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)
{
    if (attemptMove(*m_arena, dir, m_row, m_col))
    {
        if (m_arena->numberOfVampiresAt(m_row, m_col) > 0)
        {
            setDead();
            return "Player walked into a vampire and died.";
        }
        string msg = "Player moved ";
        switch (dir)
        {
          case NORTH: msg += "north"; break;
          case EAST:  msg += "east";  break;
          case SOUTH: msg += "south"; break;
          case WEST:  msg += "west";  break;
        }
        msg += ".";
        return msg;
    }
    else
        return "Player couldn't move; player stands.";
}

bool Player::isDead() const
{
    return m_dead;
}

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()
{
    for (int k = 0; k < m_nVampires; k++)
        delete m_vampires[k];
    delete m_player;
}

int Arena::rows() const
{
    return m_rows;
}

int Arena::cols() const
{
    return m_cols;
}

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

```
}

int Arena::vampireCount() const
{
    return m_nVampires;
}

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
{
    int count = 0;
    for (int k = 0; k < m_nVampires; k++)
    {
        Vampire* vp = m_vampires[k];
        if (vp->row() == r  &&  vp->col() == c)
            count++;
    }
    return count;
}

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
    for (int k = 0; k < m_nVampires; k++)
    {
        const Vampire* vp = m_vampires[k];
        char& gridChar = displayGrid[vp->row()-1][vp->col()-1];
        switch (gridChar)
        {
          case '.':  gridChar = 'V'; break;
          case 'V':  gridChar = '2'; break;
          case '9':  break;
          default:   gridChar++; break;  // '2' through '8'
        }
    }

      // 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 (m_nVampires == MAXVAMPIRES)
        return false;
    m_vampires[m_nVampires] = new Vampire(this, r, c);
    m_nVampires++;
    return true;
}

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
    for (int k = m_nVampires-1; k >= 0; k--)
    {
        Vampire* vp = m_vampires[k];
        vp->move();

        if (m_player != nullptr  &&
                vp->row() == m_player->row()  &&  vp->col() == m_player->col())
            m_player->setDead();

        if (vp->isDead())
        {
            delete vp;

              // The order of Vampire pointers in the m_vampires array is
              // irrelevant, so it's easiest to move the last pointer to
              // replace the one pointing to the now-deleted vampire.  Since
              // we are traversing the array from last to first, we know this
              // last pointer does not point to a dead vampire.

            m_vampires[k] = m_vampires[m_nVampires-1];
            m_nVampires--;
        }
    }

      // 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 (!isPosInBounds(r, c))
    {
        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)
{
    int rnew = r;
    int cnew = c;
    switch (dir)
    {
      case NORTH:  if (r <= 1)        return false; else rnew--; break;
      case EAST:   if (c >= a.cols()) return false; else cnew++; break;
      case SOUTH:  if (r >= a.rows()) return false; else rnew++; break;
      case WEST:   if (c <= 1)        return false; else cnew--; break;
    }
    r = rnew;
    c = cnew;
    return true;
}

  // 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)
{
      // How dangerous is it to stand?
    int standDanger = computeDanger(a, r, c);
```

```
      // if it's not safe, see if moving is safer
    if (standDanger > 0)
    {
        int bestMoveDanger = standDanger;
        int bestMoveDir = NORTH;  // arbitrary initialization

          // check the four directions to see if any move is
          // better than standing, and if so, record the best
        for (int dir = 0; dir < NUMDIRS; dir++)
        {
            int rnew = r;
            int cnew = c;
            if (attemptMove(a, dir, rnew, cnew))
            {
                int danger = computeDanger(a, rnew, cnew);
                if (danger < bestMoveDanger)
                {
                    bestMoveDanger = danger;
                    bestMoveDir = dir;
                }
            }
        }

          // if moving is better than standing, recommend move
        if (bestMoveDanger < standDanger)
        {
            bestDir = bestMoveDir;
            return true;
        }
    }
    return false;  // recommend standing
}

int computeDanger(const Arena& a, int r, int c)
{
      // Our measure of danger will be the number of vampires that might move
      // to position (r,c).  If a vampire is at that position, it is fatal,
      // so a large value is returned.

    if (a.numberOfVampiresAt(r,c) > 0)
        return MAXVAMPIRES+1;

    int danger = 0;
    if (r > 1)
        danger += a.numberOfVampiresAt(r-1,c);
    if (r < a.rows())
        danger += a.numberOfVampiresAt(r+1,c);
    if (c > 1)
        danger += a.numberOfVampiresAt(r,c-1);
    if (c < a.cols())
        danger += a.numberOfVampiresAt(r,c+1);

    return danger;
}

///////////////////////////////////////////////////////////////////////////
// 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
```