```
// zombies.cpp

#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 INITIAL_ZOMBIE_HEALTH = 2;

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;  // This is needed to let the compiler know that 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_health;
};

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)
 : m_arena(ap), m_row(r), m_col(c), m_health(INITIAL_ZOMBIE_HEALTH)
{
    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);
    }
}

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 not move, do not move
    int dir = randInt(0, NUMDIRS-1);  // dir is now UP, DOWN, LEFT, or RIGHT
    m_arena->determineNewPosition(m_row, m_col, dir);
}

bool Zombie::getAttacked(int dir)  // return true if dies
{
    m_health--;
    if (m_health == 0)
        return true;
    if ( ! m_arena->determineNewPosition(m_row, m_col, dir))
    {
        m_health = 0;
        return true;
    }
    return false;
}

///////////////////////////////////////////////////////////////////////////
//  Player implementations
///////////////////////////////////////////////////////////////////////////

Player::Player(Arena* ap, int r, int c)
 : m_arena(ap), m_row(r), m_col(c), m_age(0), m_dead(false)
{
    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);
    }
}

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++;
    int r = m_row;
    int c = m_col;
    if (m_arena->determineNewPosition(r, c, dir))
    {
        if (m_arena->numZombiesAt(r, c) > 0)
            m_arena->attackZombieAt(r, c, dir);
        else
        {
            m_row = r;
            m_col = c;
        }
    }
}

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

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

///////////////////////////////////////////////////////////////////////////
//  Arena implementations
///////////////////////////////////////////////////////////////////////////

Arena::Arena(int nRows, int nCols)
 : m_rows(nRows), m_cols(nCols), m_player(nullptr), m_nZombies(0)
{
    if (nRows <= 0  ||  nCols <= 0  ||  nRows > MAXROWS  ||  nCols > MAXCOLS)
    {
        cout << "***** Arena created with invalid size " << nRows << " by "
             << nCols << "!" << endl;
        exit(1);
    }
```

```
}

Arena::~Arena()
{
    for (int k = 0; k < m_nZombies; k++)
        delete m_zombies[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::zombieCount() const
{
    return m_nZombies;
}

int Arena::numZombiesAt(int r, int c) const
{
    int count = 0;
    for (int k = 0; k < m_nZombies; k++)
    {
        const Zombie* zp = m_zombies[k];
        if (zp->row() == r  &&  zp->col() == c)
            count++;
    }
    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 >= rows()) return false; else r++; break;
      case LEFT:   if (c <= 1)      return false; else c--; break;
      case RIGHT:  if (c >= cols()) return false; else c++; break;
      default:     return false;
    }
    return true;
}

void Arena::display() const
{
      // Position (row,col) of 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
    for (int k = 0; k < m_nZombies; k++)
    {
        const Zombie* zp = m_zombies[k];
        char& gridChar = grid[zp->row()-1][zp->col()-1];
        switch (gridChar)
        {
          case '.':  gridChar = 'Z'; break;
          case 'Z':  gridChar = '2'; break;
          case '9':  break;
          default:   gridChar++; break;  // '2' through '8'
        }
    }

        // Indicate player's position
    if (m_player != nullptr)
    {
          // Set the char to '@', unless there is 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)
{
      // Dynamically allocate a new Zombie and add it to the arena
    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)
{
      // Attack one zombie.  Returns true if a zombie was attacked and destroyed,
      // false otherwise (no zombie there, or the attack did not destroy the
      // zombie).
    int k = 0;
    for ( ; k < m_nZombies; k++)
    {
        if (m_zombies[k]->row() == r  &&  m_zombies[k]->col() == c)
            break;
    }
    if (k < m_nZombies  &&  m_zombies[k]->getAttacked(dir))  // zombie dies
    {
        delete m_zombies[k];
        m_zombies[k] = m_zombies[m_nZombies-1];
        m_nZombies--;
        return true;
    }
    return false;
}

bool Arena::moveZombies()
{
    for (int k = 0; k < m_nZombies; k++)
    {
        Zombie* zp = m_zombies[k];
        zp->move();
        if (zp->row() == m_player->row()  &&  zp->col() == m_player->col())
            m_player->setDead();
    }

      // return true if the player is still alive, false otherwise
    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()
{
      // Create a game
      // Use this instead to create a mini-game:   Game g(3, 4, 2);
    Game g(7, 8, 25);

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

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

// DO NOT MODIFY OR REMOVE ANY CODE BETWEEN HERE AND THE END OF THE FILE!!!
// YOU MAY MOVE TO ANOTHER FILE ALL THE CODE FROM HERE TO THE END OF FILE, BUT
// BE SURE TO MOVE *ALL* THE CODE; DON'T MODIFY OR REMOVE ANY #IFDEF, ETC.
// THE CODE IS SUITABLE FOR VISUAL C++, XCODE, AND g++ 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++

#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>
using namespace std;

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
```