**Домашнее задание.**

Выполнил: Беликов Константин  
Группа: ИУ5-36Б  
Дата: 01.12.24г.

Описание задания:

Стояла задача разработать пошаговую игру в стиле RPG

Была разработана программа, отвечающая за сражение в данной игре:

1. За них отвечал класс Arena, который мог как создавать дополнительных существ на поле боя, так и обрабатывать ход сражения.
2. В качестве существ выступало два класса, первый представлял игровых персонажей игроков, второй представлял врагов.
3. Как враги, так и персонажи могут перемещаться по полю и атаковать, пока не потратят все свои очки действия. В любой момент ход может передаться другому существу в порядке его инициативы.
4. Бой заканчивается, если hp персонажа опустились до 0.

Команды, которые способен выполнять персонаж игрока:

**m x y** – переместиться в точку с координатами (x, y).

Персонаж перемещается в данную точку самым коротким возможным путём, обходя стены и других существ. Если точка не достижима, персонаж не перемещается, если путь требует больше очков действия, чем есть у персонажа, то он перемещается на то расстояние, на которое возможно.

**a x y** – атаковать точку с координатами (x, y).

Если в точке с данными координатами нет врага или точка не в зоне действия атаки персонажа, то он не совершает действие и не тратит очки. Если после атаки hp врага опустилось до 0, то он погибает и исчезает с поля.

**t** – завершить ход.

Персонаж завершает свой ход и дожидается, пока сходят другие существа.

Код программы:

**Main.hpp**

#ifndef MAIN\_HPP

#define MAIN\_HPP

static int N = 10, M = 10;

#include <iostream>

#include <fstream>

#include <vector>

#include <queue>

#include <list>

#include <map>

#include <algorithm>

#include <cmath>

#include "Geometry.hpp"

#include "Arena.hpp"

#include "Enemy.hpp"

#include "Hero.hpp"

#include "Entity.hpp"

#endif

**main.cpp**

#include "Main.hpp"

int main() {

    int \*\*field;

    read("field.txt", field);

    Arena::Arena arena(field);

    arena.addEntity(1, {0, 0});

    arena.addEntity(2, {5, 0});

    //arena.addEntity(2, {5, 1});

    print(field);

    std::cout << "Current HP: " << arena.getHP() << std::endl;

    std::cout << std::endl << std::endl;

    while (true) {

        while(arena.type() == 2) arena.MakeMove();

        if (!arena.getHP()) break;

        print(field);

        std::cout << "Current HP: " << arena.getHP() << std::endl;

        std::cout << std::endl << std::endl;

        char key;

        int x, y;

        while(true) {

            std::cin >> key;

            if (key == 't') {arena.turn(); break;}

            std::cin >> x;

            std::cin >> y;

            switch (key)

            {

            case 'a':

                arena.Atack({x, y});

                break;

            case 'm':

                arena.move({x, y});

                print(field);

                std::cout << std::endl << std::endl;

                break;

            default:

                break;

            }

        }

    }

    std::cout << "Game Over" << std::endl;

    return 0;

}

**Arena.hpp**

#ifndef ARENA

#define ARENA

#include "Main.hpp"

#include "Entity.hpp"

#include "Hero.hpp"

#include "Enemy.hpp"

namespace Arena {

    class Arena {

        std::list<int> queue;

        std::map<int, Ent::Entity\*> entities;

        //std::map<Point, int> posId;

        int \*\*field;

        int currentId;

        int heroId;

        Hero \*currentHero;

        void changeId() {

            currentId++;

        }

        int posId(Point pos) {return field[pos.getY()][pos.getX()];}

        public:

            Arena(int \*\*&field) {

                this -> field = field;

                currentId = 0;

                heroId = -1;

                currentHero = nullptr;

            }

            ~Arena() {

                for (auto iter = queue.begin(); iter != queue.end(); iter++) {

                    delete entities[\*iter];

                    entities.erase(\*iter);

                }

            }

            void addEntity(int type, Point pos) {

                this -> changeId();

                //posId[pos] = currentId;

                switch (type)

                {

                case 1:

                    entities[currentId] = new Hero({5, 5, 5}, pos, field, currentId);

                    break;

                case 2:

                    entities[currentId] = new Enemy({4, 4, 4}, pos, field, currentId);

                    break;

                default:

                    break;

                }

                int initiative = entities[currentId] -> throwInitiative();

                if (entities[currentId] -> getType() == 1) {

                    if (heroId == -1 || initiative > entities[heroId] -> getInitiative()) {

                        heroId = currentId;

                        currentHero = dynamic\_cast<Hero\*>(entities[currentId]);

                    }

                }

                bool flag = true;

                for (auto iter = queue.begin(); iter != queue.end(); iter++) {

                    if (initiative < entities[\*iter] -> getInitiative()) {

                        flag = false;

                        queue.insert(iter, currentId);

                        break;

                     }

                }

                if (flag) queue.push\_back(currentId);

            }

            std::vector<Point> MakeMove() {

                if (entities[queue.back()] -> getType() == 1) return {};

                if (heroId == -1) {this -> turn(); return {};}

                Enemy \*enemy = dynamic\_cast<Enemy\*>(entities[queue.back()]);

                Point pos = enemy -> getPos();

                enemy -> setTarget(currentHero);

                std::vector path = enemy -> MakeMove();

                //this -> changePos(pos);

                this -> turn();

                return path;

            }

            bool Atack(Point pos) {

                if (!entities.count(posId(pos))) return false;

                int id = posId(pos);

                Ent::Entity \*target = entities[id];

                if (currentHero -> Atack(target)) {

                    if (target -> getHP() == 0) {

                        target -> fillCell(0);

                        delete target;

                        entities.erase(id);

                        field[pos.getY()][pos.getX()] = 0;

                        for (auto iter = queue.begin(); iter != queue.end(); iter++) {

                            if (\*iter == id) {

                                queue.erase(iter);

                                break;

                            }

                        }

                    }

                    return true;

                }

                return false;

            }

            std::vector<Point> move(Point pos) {

                if (this -> type() != 1) return {};

                //Point lastPos = currentHero -> getPos();

                std::vector<Point> path = currentHero -> move(pos);

                //this -> changePos(lastPos);

                return path;

            }

            int getHP() {return currentHero -> getHP();}

            void turn() {

                int id = queue.back();

                queue.pop\_back();

                queue.push\_front(id);

                id = queue.back();

                entities[id] -> recover();

                if (entities[id] -> getType() == 1) {

                    heroId = id;

                    currentHero = dynamic\_cast<Hero\*>(entities[heroId]);

                }

            }

            /\*

            bool changePos(Point pos) {

                if (!posId.count(pos)) return false;

                int id = posId[pos];

                posId.erase(pos);

                posId[entities[id] -> getPos()] = id;

                return true;

            }\*/

            int type() {return entities[queue.back()] -> getType();}

    };

}

#endif

**Entity.hpp**

#ifndef ENTITY

#define ENTITY

#include "Main.hpp"

struct Attribute {

    int str, dex, con;

};

namespace Ent {

    class Entity {

        protected:

            int type;

            int id;

            Attribute atr;

            int Max\_hp, hp;

            int Max\_points, points;

            int initiative;

            Point pos;

            int \*\*field;

            Matrix<bool> used;

            Matrix<Point> paths;

        public:

            Entity() {hp = 0;}

            Entity(Attribute atr, Point pos, int \*\*&field, int id, int type = 0) {

                this -> type = type;

                this -> id = id;

                used.resize(M, std::vector<bool> (N));

                paths.resize(M, std::vector<Point> (N));

                this -> field = field;

                this -> atr = atr;

                this -> pos = pos;

                Max\_hp = atr.con;

                hp = Max\_hp;

                initiative = 0;

                Max\_points = atr.dex;

                points = Max\_points;

                this -> fillCell(id);

            }

            bool Atack(Entity \*&target) {

            //atak statsHero

                int atackRadious = 1, atackDemage = 1, atackCost = 1;

            //

                if ((Distance(pos, target -> getPos()) <= atackRadious) && (this -> decP(atackCost))) {

                    target -> decHP(atackDemage);

                    return true;

                }

                return false;

            }

            void fillCell(int cell) {field[pos.getY()][pos.getX()] = cell;}

            void Drag(Point pos) {

                this -> fillCell(0);

                this -> pos = pos;

                this -> fillCell(id);

            }

            virtual int throwInitiative() {initiative = atr.dex; return initiative;}

            void recover() {points = Max\_points;}

            void setPos(Point pos) {this -> pos = pos;}

            int getInitiative() {return initiative;}

            Point getPos() {return pos;}

            int getHP() {return hp;}

            int getType() {return type;}

            int getId() {return id;}

            bool decP(int n = 1) {

                if (points - n < 0) return false;

                points -= n;

                return true;

            }

            void decHP(int n) {

                if (hp - n <= 0) hp = 0;

                else hp -= n;

            }

            virtual std::vector<Point> MakeMove() = 0;

    };

}

#endif

**Hero.hpp**

#ifndef HERO\_HPP

#define HERO\_HPP

#include "Main.hpp"

#include "Entity.hpp"

class Hero : public Ent::Entity{

    public:

        Hero(Attribute atr, Point pos, int \*\*&field, int id) : Entity(atr, pos, field, id, 1) {}

        std::vector<Point> move(Point finish) {

            if (short\_path(field, used, paths, pos, finish)) return {};

            std::vector<Point> path;

            for (Point point : make\_path(paths, pos, finish)) {

                if (!(this -> decP())) break;

                path.push\_back(point);

            }

            if (!path.empty()) this -> Drag(path.back());

            return path;

        }

        std::vector<Point> MakeMove() override {return {};}

};

#endif

**Enemy.hpp**

#ifndef ENEMY\_HPP

#define ENEMY\_HPP

#include "Main.hpp"

#include "Hero.hpp"

#include "Entity.hpp"

class Enemy : public Ent::Entity{

    Matrix<int> dist;

    Entity \*target;

    public:

        Enemy(Attribute atr, Point pos, int \*\*&field, int id) : Entity(atr, pos, field, id, 2) {

            dist.resize(M, std::vector<int> (N));

            target = nullptr;

            }

        void setTarget(Hero \*&target) {this -> target = target;}

        std::vector<Point> MakeMove() override {

            std::vector<Point> path, circle;

            Point min\_p;

            int min\_dist = (M \* N);

            int x, y;

            int R = 1;

            make\_way\_matrix(field, used, paths, dist, pos);

            //try going very close

            do {

                circle = make\_circle(target -> getPos(), R++);

                for (Point p : circle) {

                    p.fill(x, y);

                    if (dist[y][x] == -1) continue;

                    if (dist[y][x] < min\_dist) {min\_p = p; min\_dist = dist[y][x];};

                }

                if (!min\_p.is\_none() && (this -> decP(min\_dist))) {

                    path = make\_path(paths, pos, min\_p);

                    this -> Drag(min\_p);

                    break;

                }

            } while (!circle.empty());

            //atack

            while (this -> Atack(target)) {std::cout << "atacking" << std::endl;}

            return path;

        }

};

#endif

**Geometry.hpp**

#ifndef GEOMETRY\_HPP

#define GEOMETRY\_HPP

template<typename T>

using Matrix = std::vector<std::vector<T>>;

class Point {

    int x, y;

    public:

        void set(int x, int y) {

            this -> x = x;

            this -> y = y;

        }

        Point() {x = -1; y = -1;};

        Point(int x, int y) {

            this -> set(x, y);

        }

        int getX() {return x;}

        int getY() {return y;}

        std::vector<int> get() {return {x, y};}

        void fill(int &x, int &y) {

            x = this -> x;

            y = this -> y;

        }

        bool is\_none() {

            if ((x == -1) && (y == -1)) return true;

            return false;

        }

        bool operator ==(Point p) {return (x == p.x) && (y == p.y);}

        bool operator ==(std::vector<int> v) {return (x == v[0]) && (y == v[1]);}

        bool operator <(const Point &p) const {return (x + y) < (p.x + p.y);}

        bool operator !=(Point p) {return (x != p.x) || (y != p.y);}

        Point operator +(Point& p) {return Point(x + p.x, y + p.y);}

        Point operator -(Point& p) {return Point(x - p.x, y - p.y);}

        //Point& operator =(const Point &p) {return \*this;}

        void operator =(Point p) {this -> set(p.x, p.y);}

};

int read(const char\* file\_name, int \*\*&arr);

void print(int\*\* arr);

double Distance(Point p1, Point p2);

int F(Point p, Point p1, Point p2);

std::vector<Point> draw\_line(Point p1, Point p2);

//std::vector<Point> short\_path(int field [M][N], Point start, Point finish);

int short\_path(int \*\*field, Matrix<bool> &used, Matrix<Point> &paths, Point start, Point finish);

std::vector<Point> make\_path(Matrix<Point> &paths, Point p1, Point p2);

void make\_way\_matrix(int\*\* field, Matrix<bool> &used, Matrix<Point> &paths, Matrix<int> &dist, Point pos, int points = (N \* M));

std::vector<Point> make\_circle(Point O, int R);

std::vector<Point> make\_circle2(Point O, int R);

std::vector<Point> remove(const std::vector<Point> &v1);

#endif

**Geometry.cpp**

#include "Main.hpp"

int read(const char\* file\_name, int \*\*&arr) {

    std::ifstream fin(file\_name, std::ios::in);

    if (!fin) {

        std::cout << "Heт файла " << file\_name << std::endl;

        return -1;

    }

    arr = new int\* [M];

    for (int i = 0; i < M; i++) {

        arr[i] = new int [N];

        for (int j = 0; j < N; j++) {

            fin >> arr[i][j];

        }

    }

    fin.close();

    return 0;

}

void print(int\*\* arr) {

    for (int i = 0; i < M; i++) {

        for (int j = 0; j < N; j++) {

            std::cout << arr[i][j] << ' ';

        }

        std::cout << std::endl;

    }

}

int F(Point p, Point p1, Point p2) {

    p1 = p1 - p;

    p2 = p2 - p;

    int xy1 = abs(p1.getX() \* p2.getY());

    int xy2 = abs(p2.getX() \* p1.getY());

    return abs(xy1 - xy2);

}

double Distance(Point p1, Point p2) {

    int x1, y1, x2, y2;

    p1.fill(x1, y1);

    p2.fill(x2, y2);

    return pow(pow(x1 - x2, 2) + pow(y1 - y2, 2), 0.5);

}

std::vector<Point> draw\_line(Point p1, Point p2) {

    std::vector<Point> path;

    path.push\_back(p1);

    Point pk1{0, 0}, pk2{0, 0};

    int dx = p2.getX() - p1.getX();

    int dy = p2.getY() - p1.getY();

    if (dx != 0) pk1.set(dx / abs(dx), 0);

    if (dy != 0) pk2.set(0, dy / abs(dy));

    if (!dx || !dy) {

        while (p1 != p2) {

            p1 = p1 + pk1 + pk2;

            path.push\_back(p1);

        }

    } else {

        Point p = p1;

        while (p != p2) {

            p = p + (F(p1, p2, p + pk1) < F(p1, p2, p + pk2) ? pk1 : pk2);

            path.push\_back(p);

        }

    }

    return path;

}

int short\_path(int \*\*field, Matrix<bool> &used, Matrix<Point> &paths, Point start, Point finish) {

    for (int i = 0; i < N; ++i) {

        for (int j = 0; j < M; ++j) used[j][i] = false;

    }

    Point to;

    std::queue<Point> q;

    used[start.getY()][start.getX()] = true;

    q.push(start);

    int X, Y, x, y;

    while (!q.empty()) {

            Point v = q.front();

            q.pop();

            X = v.getX(); Y = v.getY();

            for (int i = 0; i < 4; ++i) {

                switch (i)

                {

                case 0:

                    x = X + 1; y = Y;

                    break;

                case 1:

                    x = X - 1; y = Y;

                    break;

                case 2:

                    x = X; y = Y + 1;

                    break;

                case 3:

                    x = X; y = Y - 1;

                    break;

                }

                if ((x < 0 || x >= N) || (y < 0 || y >= M) || (field[y][x] != 0)) continue;

                to.set(x, y);

                if (to == finish) {

                    paths[y][x] = v;

                    return 0;

                }

                if (!used[y][x]) {

                    used[y][x] = true;

                    q.push (to);

                    paths[y][x].set(X, Y);

                }

            }

    }

    //std::cout << "no such way" << std::endl;

    return -1;

}

void make\_way\_matrix(int\*\* field, Matrix<bool> &used, Matrix<Point> &paths, Matrix<int> &dist, Point pos, int points) {

    for (int i = 0; i < N; ++i) {

        for (int j = 0; j < M; ++j) {

            used[j][i] = false;

            dist[j][i] = -1;

        };

    }

    Point to;

    std::queue<Point> q;

    used[pos.getY()][pos.getX()] = true;

    dist[pos.getY()][pos.getX()] = 0;

    q.push(pos);

    int X, Y, x, y;

    while (!q.empty()) {

            Point v = q.front();

            q.pop();

            X = v.getX(); Y = v.getY();

            if (dist[Y][X] == points) continue;

            for (int i = 0; i < 4; ++i) {

                switch (i)

                {

                case 0:

                    x = X + 1; y = Y;

                    break;

                case 1:

                    x = X - 1; y = Y;

                    break;

                case 2:

                    x = X; y = Y + 1;

                    break;

                case 3:

                    x = X; y = Y - 1;

                    break;

                }

                if ((x < 0 || x >= N) || (y < 0 || y >= M) || (field[y][x] != 0)) continue;

                to.set(x, y);

                if (!used[y][x]) {

                    used[y][x] = true;

                    q.push (to);

                    paths[y][x].set(X, Y);

                    dist[y][x] = dist[Y][X] + 1;

                }

            }

    }

    return;

}

std::vector<Point> make\_path(Matrix<Point> &paths, Point p1, Point p2) {

    std::vector<Point> path;

    while (!p2.is\_none()) {

        if (p1 == p2) {

            std::reverse(path.begin(), path.end());

            return path;

        }

        path.push\_back(p2);

        p2 = paths[p2.getY()][p2.getX()];

    }

    return {};

}

std::vector<Point> make\_circle(Point O, int R) {

    std::vector<Point> circle (4 \* R);

    int k[4][2] = {{1, 1}, {-1, 1}, {-1 , -1}, {1, -1}};

    const int cx = O.getX(), cy = O.getY();

    int x, y;

    for (int i = 0; i < 4; i++) {

        for (int j = 0; j < R; j++) {

            x = j \* k[i][0];

            y = (R - j) \* k[i][1];

            if (i % 2 == 1) std::swap(x, y);

            x += cx; y += cy;

            circle[i \* R + j].set(x, y);

        }

    }

    return remove(circle);

}

std::vector<Point> make\_circle2(Point O, int R) {

    std::vector<Point> circle;

    const int cx = O.getX(), cy = O.getY();

    int x = 0, y = R;

    int r, r1, r2, r3;

    R \*= R;

    while (y > 0) {

        r = x \* x + y \* y;

        r1 = r + 2 \* x + 1;

        r2 = r + 2 \* x - 2 \* y + 2;

        r3 = r - 2 \* y + 1;

        if (r1 <= R) {x++; r = r1;}

        else if (r2 <= R) {x++; y--; r = r2;}

        else {y--; r = r3;}

        circle.push\_back({cx + x, cy + y});

        circle.push\_back({cx + y, cy - x});

        circle.push\_back({cx - x, cy - y});

        circle.push\_back({cx - y, cy + x});

    }

    return remove(circle);

}

std::vector<Point> remove(const std::vector<Point> &v1) {

    std::vector<Point> v2;

    int x, y;

    for (Point p : v1) {

        x = p.getX(); y = p.getY();

        if (p.is\_none() || (x < 0 || x >= N) || (y < 0 || y >= M)) continue;

        v2.push\_back(p);

    }

    return v2;

}

/\*

double H(Point p, Point p1, Point p2) {

    p1 = p1 - p;

    p2 = p2 - p;

    int x1 = abs(p1.getX()), y1 = abs(p1.getY());

    int x2 = abs(p2.getX()), y2 = abs(p2.getY());

    double pol1, pol2, pol3;

    pol1 = pow(x1, 2) + pow(y1, 2);

    pol2 = pow(x2, 2) + pow(y2, 2);

    pol3 = x1 \* x2 + y1 \* y2;

    if (!pol1) return pol2;

    return pow(pol2 - pow(pol3, 2) / pol1, 0.5);

}

\*/

**field.txt**

**0 0 0 0 0 0 0 0 0 0**

**0 0 0 0 0 0 0 0 0 0**

**0 0 0 0 8 0 0 0 0 0**

**0 0 0 0 8 0 0 0 0 0**

**0 0 0 0 8 0 0 0 0 0**

**0 0 0 0 0 0 0 0 0 0**

**0 0 0 0 0 0 0 0 0 0**

**0 0 0 0 0 0 0 0 0 0**

**0 0 0 0 0 0 0 0 0 0**

**0 0 0 0 0 0 0 0 0 0**

Снимки экрана:

1 – Персонаж

2 – Враг

0 – Пустая клетка

8 - Стена

1 0 0 0 0 2 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

Current HP: 5

m 0 5

0 0 0 0 0 2 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

1 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

m 0 6

0 0 0 0 0 2 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

1 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

t

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 0 0 0 8 2 0 0 0 0

1 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

Current HP: 5

m 1 3

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 1 0 0 8 0 0 0 0 0

0 0 0 0 8 2 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

t

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 1 0 0 8 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 0 2 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

Current HP: 5

t

atacking

atacking

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 1 0 0 8 0 0 0 0 0

0 2 0 0 8 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

Current HP: 3

a 1 4

a 1 4

a 1 4

a 1 4

t

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 1 0 0 8 0 0 0 0 0

0 0 0 0 8 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

Current HP: 3