Министерство образования и науки Кыргызской Республики

Кыргызский государственный технический университет

им. И.Раззакова

Институт информационных технологий

Кафедра «Программное обеспечение компьютерных систем»

Направление: «Программная инженерия»

Дисциплина: «Объектно-Ориентированное Программирование»

**Курсовой проект**

**Тема: игра «Кураж(будь тише!)»**

Выполнил: студент группы ПИ-2-21

Закиров Мустафа

Проверил: Мусабаев Э.Б

**Бишкек 2025**

### Введение

Программа представляет собой консольную игру в жанре roguelike, где игрок управляет персонажем, перемещаясь по лабиринту и избегая врагов. Игровой процесс вдохновлен механикой Crypt of the NecroDancer: за один ход игрока все враги также делают один ход. Цель игры — пройти через три уровня разной сложности (маленький, средний и большой), избегая столкновений с врагами, и достичь выхода, обозначенного символом E. После прохождения всех уровней игрок получает виртуальный кубок победы.

Игра реализована с использованием объектно-ориентированного программирования (ООП) на языке C++. В ней представлены различные классы: персонаж игрока (Player), враги (Enemy, PatrolEnemy, ChaseEnemy), карта (Map), а также вспомогательные классы для управления вводом, отрисовкой и уровнями. Каждый уровень представляет собой лабиринт с уникальной структурой, где игрок должен учитывать механику шума (!), привлекающую врагов. Враги делятся на патрулирующих и преследующих, каждый с собственной логикой движения.

### Описание классов

#### Класс Position

**Назначение:** Класс Position предназначен для хранения и управления координатами объектов на карте. Он используется всеми игровыми объектами для определения их положения.  
**Поля:**

* x (положение по X): Целочисленное поле, обозначающее горизонтальную координату.
* y (положение по Y): Целочисленное поле, обозначающее вертикальную координату.  
  **Методы:**
* Position(int x\_ = 0, int y\_ = 0): Конструктор, инициализирующий координаты значениями по умолчанию (0, 0).
* bool operator==(const Position& other) const: Оператор сравнения двух позиций.
* int distanceTo(const Position& other) const: Метод, вычисляющий манхэттенское расстояние до другой позиции.

#### Класс GameObject

**Назначение:** Абстрактный базовый класс для всех игровых объектов, предоставляющий общие свойства и методы.  
**Поля:**

* pos (позиция): Объект класса Position, хранящий координаты объекта.
* lastMove (последнее направление): Символьное поле, хранящее стрелку последнего движения (>, <, v, ^).  
  **Методы:**
* GameObject(int x, int y): Конструктор, инициализирующий позицию объекта.
* virtual void update() = 0: Чисто виртуальный метод для обновления состояния объекта.
* virtual char getSymbol() const = 0: Чисто виртуальный метод, возвращающий символ объекта.
* Position getPosition() const: Возвращает текущую позицию.
* void setPosition(Position newPos, char direction): Устанавливает новую позицию и направление движения.
* char getLastMove() const: Возвращает символ последнего направления.

#### Класс Map

**Назначение:** Класс Map отвечает за представление игровой карты, включая лабиринт, стены, выход и шум.  
**Поля:**

* grid (сетка): Вектор векторов символов, представляющий карту.
* noiseTimer (таймер шума): Вектор векторов целых чисел, отслеживающий время жизни шума.
* width (ширина): Целочисленное поле, задающее ширину карты.
* height (высота): Целочисленное поле, задающее высоту карты.  
  **Методы:**
* Map(const string& filename, int w, int h): Конструктор, загружающий карту из файла или создающий базовую структуру.
* bool isWalkable(Position pos) const: Проверяет, проходима ли клетка.
* bool isExit(Position pos) const: Проверяет, является ли клетка выходом.
* void setSymbol(Position pos, char symbol): Устанавливает символ на карте.
* char getSymbol(Position pos) const: Возвращает символ в заданной позиции.
* void updateNoise(): Обновляет состояние шума, уменьшая таймер и убирая его при истечении.
* int getWidth() const, int getHeight() const: Возвращают размеры карты.

#### Класс Player

**Назначение:** Класс Player представляет персонажа игрока, управляемого через WASD, с механикой шума и здоровья.  
**Поля:**

* health (здоровье): Целочисленное поле, обозначающее количество здоровья (по умолчанию 3).  
  **Методы:**
* Player(int x, int y): Конструктор, устанавливающий начальную позицию и здоровье.
* void update() override: Пустая реализация метода обновления.
* void move(int dx, int dy, Map& map): Перемещает игрока и создает шум при движении.
* void makeNoise(Map& map): Создает шум (!) в радиусе 2 клеток, учитывая стены.
* char getSymbol() const override: Возвращает символ @.
* int getHealth() const: Возвращает текущее здоровье.
* void takeDamage(): Уменьшает здоровье на 1.

#### Класс Enemy

**Назначение:** Абстрактный базовый класс для врагов, определяющий общие свойства и поведение.  
**Поля:**

* symbol (символ): Символьное поле, обозначающее отображаемый символ врага.
* chasing (преследование): Булево поле, указывающее, преследует ли враг игрока.  
  **Методы:**
* Enemy(int x, int y, char sym): Конструктор, задающий начальную позицию и символ.
* char getSymbol() const override: Возвращает символ врага.
* virtual void hearNoise(const Map& map, const Position& playerPos): Проверяет шум в радиусе 3 клеток и активирует преследование.

#### Класс PatrolEnemy

**Назначение:** Представляет патрулирующих врагов, движущихся по заданному маршруту, пока не услышат шум.  
**Поля:**

* patrolRoute (маршрут): Вектор позиций, задающий маршрут патрулирования.
* routeIndex (индекс маршрута): Целочисленное поле, указывающее текущую точку маршрута.
* target (цель): Указатель на игрока для преследования.
* map (карта): Указатель на карту для проверки стен.  
  **Методы:**
* PatrolEnemy(int x, int y, const Player\* player, const Map\* m, int routeType): Конструктор с выбором типа маршрута.
* void update() override: Обновляет движение: патрулирование по маршруту или преследование игрока.

#### Класс ChaseEnemy

**Назначение:** Представляет преследующих врагов, активирующихся только при шуме и движущихся к игроку.  
**Поля:**

* target (цель): Указатель на игрока.
* map (карта): Указатель на карту для проверки стен.  
  **Методы:**
* ChaseEnemy(int x, int y, const Player\* player, const Map\* m): Конструктор, задающий начальные параметры.
* void update() override: Перемещает врага к игроку, если активировано преследование.

#### Класс InputHandler

**Назначение:** Класс для обработки ввода пользователя (WASD) и управления игроком.  
**Методы:**

* void handleInput(Player& player, Map& map): Обрабатывает ввод и вызывает движение игрока.

#### Класс Renderer

**Назначение:** Класс для отрисовки игрового поля и интерфейса в консоли.  
**Методы:**

* void render(const Map& map, const Player& player, const vector<Enemy\*>& enemies, int levelNum): Отображает карту, объекты и легенду.
* void renderVictory(): Отображает экран победы с кубком.

#### Класс Level

**Назначение:** Представляет игровой уровень, содержащий карту, игрока и врагов.  
**Поля:**

* map (карта): Объект класса Map.
* player (игрок): Указатель на игрока.
* enemies (враги): Вектор указателей на врагов.

**Методы:**

* Level(const string& mapFile, int startX, int startY, int width, int height): Конструктор, инициализирующий уровень с заданными размерами.
* void update(): Обновляет состояние уровня (шум и врагов).
* Player\* getPlayer(), const vector<Enemy\*>& getEnemies() const, Map& getMap(): Методы доступа к данным уровня.
* ~Level(): Деструктор, освобождающий память.

#### Класс Game:

**Назначение:** Управляет всей игровой сессией, включая уровни и их последовательность.  
**Поля:**

* levels (уровни): Вектор указателей на уровни.
* currentLevelIndex (текущий уровень): Индекс активного уровня.
* inputHandler (обработчик ввода): Объект класса InputHandler.
* renderer (отрисовщик): Объект класса Renderer.
* running (запущена ли игра): Булево поле, указывающее состояние игры.  
  **Методы:**
* Game(): Конструктор, создающий три уровня разного размера.
* void run(): Основной цикл игры, управляющий уровнями и победой.
* ~Game(): Деструктор, освобождающий память.

# **Диаграмма классов**

Figure 1 Диаграмма классов

# **Листинг программы**

#include <iostream>

#include <vector>

#include <fstream>

#include <conio.h>

#include <cstdlib>

#include <string>

using namespace std;

// Класс для координат

class Position {

public:

int x, y;

Position(int x\_ = 0, int y\_ = 0) : x(x\_), y(y\_) {}

bool operator==(const Position& other) const { return x == other.x && y == other.y; }

int distanceTo(const Position& other) const { return abs(x - other.x) + abs(y - other.y); }

};

// Абстрактный класс для объектов на карте

class GameObject {

protected:

Position pos;

char lastMove;

public:

GameObject(int x, int y) : pos(x, y), lastMove(' ') {}

virtual void update() = 0;

virtual char getSymbol() const = 0;

Position getPosition() const { return pos; }

void setPosition(Position newPos, char direction) {

pos = newPos;

lastMove = direction;

}

char getLastMove() const { return lastMove; }

};

// Класс карты

class Map {

private:

vector<vector<char>> grid;

vector<vector<int>> noiseTimer;

int width, height;

public:

Map(const string& filename, int w, int h) : width(w), height(h) {

grid.resize(height, vector<char>(width, '.'));

noiseTimer.resize(height, vector<int>(width, 0));

ifstream file(filename);

if (file.is\_open()) {

for (int y = 0; y < height && !file.eof(); y++) {

string line;

getline(file, line);

for (int x = 0; x < min(width, (int)line.size()); x++) {

grid[y][x] = line[x];

}

}

file.close();

}

else {

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

grid[0][i] = '#';

grid[height - 1][i] = '#';

}

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

grid[i][0] = '#';

grid[i][width - 1] = '#';

}

}

}

bool isWalkable(Position pos) const {

return pos.x >= 0 && pos.x < width && pos.y >= 0 && pos.y < height && grid[pos.y][pos.x] != '#';

}

bool isExit(Position pos) const {

return pos.x >= 0 && pos.x < width && pos.y >= 0 && pos.y < height && grid[pos.y][pos.x] == 'E';

}

void setSymbol(Position pos, char symbol) {

grid[pos.y][pos.x] = symbol;

if (symbol == '!') noiseTimer[pos.y][pos.x] = 2;

}

char getSymbol(Position pos) const { return grid[pos.y][pos.x]; }

void updateNoise() {

for (int y = 0; y < height; y++) {

for (int x = 0; x < width; x++) {

if (grid[y][x] == '!' && noiseTimer[y][x] > 0) {

noiseTimer[y][x]--;

if (noiseTimer[y][x] == 0) grid[y][x] = '.';

}

}

}

}

int getWidth() const { return width; }

int getHeight() const { return height; }

};

// Класс игрока

class Player : public GameObject {

private:

int health;

public:

Player(int x, int y) : GameObject(x, y), health(3) {}

void update() override {}

void move(int dx, int dy, Map& map) {

Position newPos(pos.x + dx, pos.y + dy);

if (map.isWalkable(newPos)) {

char direction = ' ';

if (dx > 0) direction = '>';

else if (dx < 0) direction = '<';

else if (dy > 0) direction = 'v';

else if (dy < 0) direction = '^';

setPosition(newPos, direction);

makeNoise(map);

}

}

void makeNoise(Map& map) {

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

int dx = (dir == 0 ? 1 : dir == 1 ? -1 : 0);

int dy = (dir == 2 ? 1 : dir == 3 ? -1 : 0);

for (int i = 1; i <= 2; i++) {

Position noisePos(pos.x + dx \* i, pos.y + dy \* i);

if (!map.isWalkable(noisePos)) break;

if (map.getSymbol(noisePos) != 'E') {

map.setSymbol(noisePos, '!');

}

}

}

}

char getSymbol() const override { return '@'; }

int getHealth() const { return health; }

void takeDamage() { if (health > 0) health--; }

};

// Базовый класс врага

class Enemy : public GameObject {

protected:

char symbol;

bool chasing;

public:

Enemy(int x, int y, char sym) : GameObject(x, y), symbol(sym), chasing(false) {}

char getSymbol() const override { return symbol; }

virtual void hearNoise(const Map& map, const Position& playerPos) {

for (int dy = -3; dy <= 3; dy++) {

for (int dx = -3; dx <= 3; dx++) {

Position checkPos(pos.x + dx, pos.y + dy);

if (checkPos.x >= 0 && checkPos.x < map.getWidth() && checkPos.y >= 0 && checkPos.y < map.getHeight()) {

if (map.getSymbol(checkPos) == '!') {

chasing = true;

return;

}

}

}

}

}

};

// Враг, патрулирующий по маршруту

class PatrolEnemy : public Enemy {

private:

vector<Position> patrolRoute;

int routeIndex;

const Player\* target;

const Map\* map;

public:

PatrolEnemy(int x, int y, const Player\* player, const Map\* m, int routeType = 0)

: Enemy(x, y, 'P'), routeIndex(0), target(player), map(m) {

if (routeType == 0) {

patrolRoute = { Position(x, y), Position(x + 1, y), Position(x + 2, y), Position(x + 1, y + 1), Position(x, y + 1) };

}

else if (routeType == 1) {

patrolRoute = { Position(x, y), Position(x, y + 1), Position(x + 1, y + 1), Position(x + 1, y), Position(x, y) };

}

else {

patrolRoute = { Position(x, y), Position(x + 1, y), Position(x + 2, y), Position(x + 3, y), Position(x + 3, y + 1) };

}

}

void update() override {

if (chasing) {

Position playerPos = target->getPosition();

int dx = playerPos.x - pos.x;

int dy = playerPos.y - pos.y;

Position newPos = pos;

char direction = ' ';

if (abs(dx) > abs(dy)) {

newPos.x += (dx > 0 ? 1 : -1);

direction = (dx > 0 ? '>' : '<');

}

else {

newPos.y += (dy > 0 ? 1 : -1);

direction = (dy > 0 ? 'v' : '^');

}

if (map->isWalkable(newPos)) setPosition(newPos, direction);

}

else {

routeIndex = (routeIndex + 1) % patrolRoute.size();

Position newPos = patrolRoute[routeIndex];

int dx = newPos.x - pos.x;

int dy = newPos.y - pos.y;

char direction = ' ';

if (dx > 0) direction = '>';

else if (dx < 0) direction = '<';

else if (dy > 0) direction = 'v';

else if (dy < 0) direction = '^';

if (map->isWalkable(newPos)) setPosition(newPos, direction);

}

}

};

// Враг, преследующий игрока

class ChaseEnemy : public Enemy {

private:

const Player\* target;

const Map\* map;

public:

ChaseEnemy(int x, int y, const Player\* player, const Map\* m) : Enemy(x, y, 'C'), target(player), map(m) {}

void update() override {

if (!chasing) return;

Position playerPos = target->getPosition();

int dx = playerPos.x - pos.x;

int dy = playerPos.y - pos.y;

Position newPos = pos;

char direction = ' ';

if (abs(dx) > abs(dy)) {

newPos.x += (dx > 0 ? 1 : -1);

direction = (dx > 0 ? '>' : '<');

}

else {

newPos.y += (dy > 0 ? 1 : -1);

direction = (dy > 0 ? 'v' : '^');

}

if (map->isWalkable(newPos)) setPosition(newPos, direction);

}

};

// Класс обработки ввода

class InputHandler {

public:

void handleInput(Player& player, Map& map) {

char input = \_getch();

switch (input) {

case 'w': player.move(0, -1, map); break;

case 's': player.move(0, 1, map); break;

case 'a': player.move(-1, 0, map); break;

case 'd': player.move(1, 0, map); break;

}

}

};

// Класс отрисовки

class Renderer {

public:

void render(const Map& map, const Player& player, const vector<Enemy\*>& enemies, int levelNum) {

system("cls");

for (int y = 0; y < map.getHeight(); y++) {

for (int x = 0; x < map.getWidth(); x++) {

Position pos(x, y);

bool objectRendered = false;

if (pos == player.getPosition()) {

cout << player.getLastMove() << player.getSymbol();

objectRendered = true;

}

else {

for (const auto\* enemy : enemies) {

if (pos == enemy->getPosition()) {

cout << enemy->getLastMove() << enemy->getSymbol();

objectRendered = true;

break;

}

}

}

if (!objectRendered) cout << " " << map.getSymbol(pos);

}

cout << endl;

}

cout << "Level: " << levelNum + 1 << " | Health: " << player.getHealth() << endl;

cout << "Legend: @ - Player, P - Patrol Enemy, C - Chase Enemy, # - Wall, E - Exit, ! - Noise, Arrows - Last Move" << endl;

}

void renderVictory() {

system("cls");

cout << endl << endl;

cout << " Congratulations! You Won!" << endl << endl;

cout << " \_\_\_\_\_\_\_" << endl;

cout << " / \\" << endl;

cout << " /\_\_\_\_\_\_\_\_\_\\" << endl;

cout << " | TROPHY |" << endl;

cout << " |\_\_\_\_\_\_\_\_\_|" << endl << endl;

cout << "Press any key to exit..." << endl;

\_getch();

}

};

// Класс уровня

class Level {

private:

Map map;

Player\* player;

vector<Enemy\*> enemies;

public:

Level(const string& mapFile, int startX, int startY, int width, int height) : map(mapFile, width, height) {

player = new Player(startX, startY);

if (width == 15) { // Level 1

enemies.push\_back(new PatrolEnemy(5, 5, player, &map, 0));

enemies.push\_back(new ChaseEnemy(10, 5, player, &map));

}

else if (width == 25) { // Level 2

enemies.push\_back(new PatrolEnemy(10, 5, player, &map, 0));

enemies.push\_back(new PatrolEnemy(15, 10, player, &map, 1));

enemies.push\_back(new ChaseEnemy(20, 5, player, &map));

}

else { // Level 3

enemies.push\_back(new PatrolEnemy(10, 10, player, &map, 0));

enemies.push\_back(new PatrolEnemy(15, 15, player, &map, 1));

enemies.push\_back(new PatrolEnemy(5, 20, player, &map, 2));

enemies.push\_back(new ChaseEnemy(20, 15, player, &map));

enemies.push\_back(new ChaseEnemy(25, 10, player, &map));

}

}

void update() {

map.updateNoise();

for (auto\* enemy : enemies) {

enemy->hearNoise(map, player->getPosition());

enemy->update();

if (enemy->getPosition() == player->getPosition()) {

player->takeDamage();

}

}

}

Player\* getPlayer() { return player; }

const vector<Enemy\*>& getEnemies() const { return enemies; }

Map& getMap() { return map; }

~Level() {

delete player;

for (auto\* enemy : enemies) delete enemy;

}

};

// Главный класс игры

class Game {

private:

vector<Level\*> levels;

int currentLevelIndex;

InputHandler inputHandler;

Renderer renderer;

bool running;

public:

Game() : currentLevelIndex(0), running(true) {

levels.push\_back(new Level("level1.txt", 1, 1, 15, 10)); // Маленький

levels.push\_back(new Level("level2.txt", 1, 1, 25, 15)); // Средний

levels.push\_back(new Level("level3.txt", 1, 1, 40, 25)); // Большой

}

void run() {

while (running) {

Level\* currentLevel = levels[currentLevelIndex];

renderer.render(currentLevel->getMap(), \*currentLevel->getPlayer(), currentLevel->getEnemies(), currentLevelIndex);

if (currentLevel->getPlayer()->getHealth() <= 0) {

cout << "Game Over!" << endl;

running = false;

break;

}

inputHandler.handleInput(\*currentLevel->getPlayer(), currentLevel->getMap());

currentLevel->update();

if (currentLevel->getMap().isExit(currentLevel->getPlayer()->getPosition())) {

if (currentLevelIndex < levels.size() - 1) {

currentLevelIndex++;

cout << "Moving to next level..." << endl;

\_getch();

}

else {

renderer.renderVictory();

running = false;

}

}

}

}

~Game() {

for (auto\* level : levels) delete level;

}

};

int main() {

Game game;

game.run();

return 0;

}

Описание уровней

#### Уровень 1: Маленький (10x10)

Первый уровень представляет собой небольшую комнату размером 10x10 клеток. Это вводный этап, где игрок знакомится с основными механиками. В комнате появляется один патрулирующий враг (P), движущийся по простому маршруту, и один предмет — зелье здоровья (I). Выход (E) находится в правом верхнем углу. Лабиринт минималистичен, с несколькими стенами для создания укрытий. Цель — подобрать предмет и достичь выхода, избегая врага.

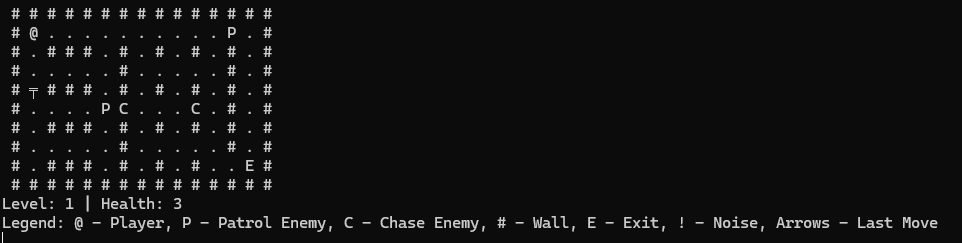
Скрин

Figure 2 первый уровень

#### Уровень 2: Средний (25x15)

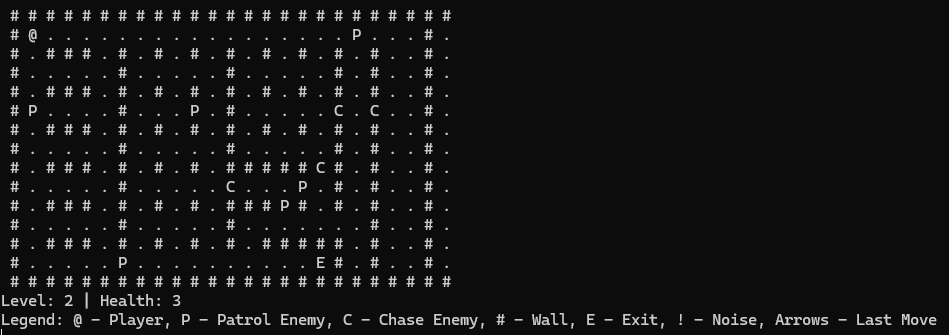
Второй уровень увеличивает сложность, представляя комнату размером 25x15 клеток. Здесь появляются два патрулирующих врага (P) с разными маршрутами и один преследующий враг (C), активирующийся при шуме. Лабиринт становится более запутанным, с длинными коридорами и тупиками. Предмет (I) спрятан в одном из тупиков, а выход (E) расположен в правом нижнем углу. Игроку нужно планировать свои действия, чтобы избежать ловушек и привлечения врагов.

Figure 3 Уровень 2

#### Уровень 3: Большой (40x25)

Третий уровень — самый сложный, с размером 40x25 клеток. Это обширный лабиринт с множеством стен, открытых зон и пересекающихся путей. В комнате находятся три патрулирующих врага (P) с уникальными маршрутами и два преследующих врага (C). Предмет (I) расположен в центре лабиринта, а выход (E) — в правом нижнем углу. Игроку требуется стратегически использовать шум и укрытия, чтобы пройти уровень и завершить игру, после чего отображается экран победы с кубком.

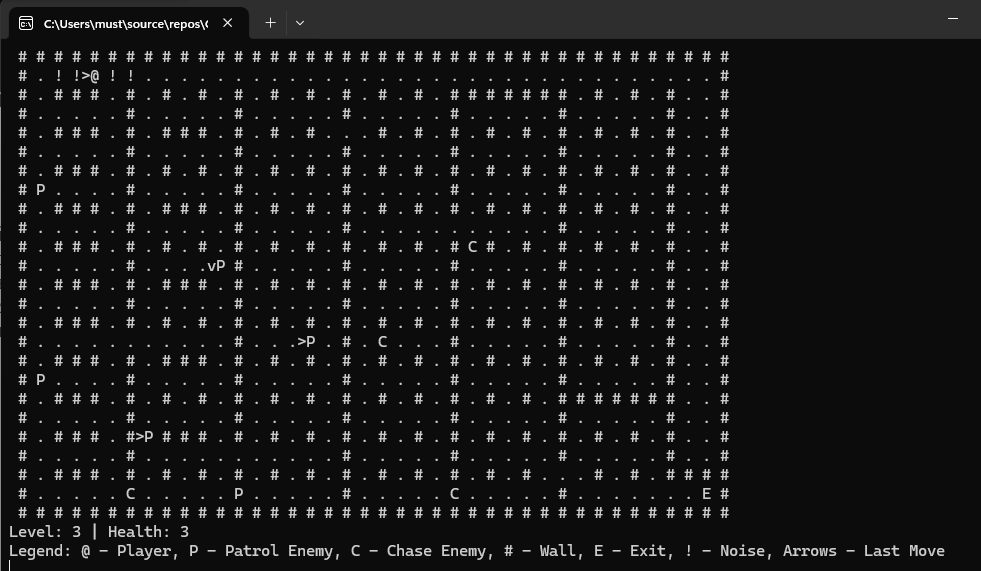


Figure 3 Уровень 3



Figure 5 победа

**Заключение**

В процессе разработки консольной игры-лабиринта я углубил свои знания объектно-ориентированного программирования на C++. Использование классов и наследования позволило эффективно организовать структуру игры, разделив функциональность между базовыми (GameObject, Enemy) и производными классами (Player, PatrolEnemy, ChaseEnemy, Boss). Виртуальные методы обеспечили полиморфизм, позволяя различным объектам (игрок, враги, босс) вести себя по-разному в зависимости от их типа.

Работа с контейнерами STL, такими как vector для хранения врагов, предметов и уровней, показала важность управления данными в динамических системах. Реализация указателей и динамического выделения памяти обеспечила гибкость при создании и удалении объектов, особенно в классе Level. Добавление меню, инвентаря и механик атаки углубило понимание взаимодействия между классами и обработки пользовательского ввода.

Этот проект стал ценным опытом, который демонстрирует применение ООП для создания интерактивных приложений. Полученные навыки будут полезны для разработки более сложных программных продуктов в будущем.

GitHub: <https://github.com/Ab1eman/OOP_game_term>