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 «Carnage»**

Выполнил: студент группы ПИ-3-21

Копжашаров Азамат

Проверил: Мусабаев Э.Б

**Бишкек 2024**

Оглавление

# **Введение**

Программа представляет собой рогалик (игру жанра roguelike), где игрок управляет персонажем и сражается с врагами. В игре есть классы персонажей и врагов, каждый из которых обладает своими уникальными характеристиками и способностями. Персонаж игрока представлен классом Character, враги – классом Enemy, а финальный босс – классом Boss.

Игра проходит в комнате размером 10x10 клеток, где персонаж может перемещаться, сражаться с врагами и собирать предметы. Враги и боссы также имеют свои собственные алгоритмы передвижения и атаки, добавляя динамику и разнообразие в игровой процесс.

# **Описание классов**

## **Класс GameObject**

**Назначение**: Класс GameObject предназначен для представления базового игрового объекта в игре. Он предоставляет основные характеристики и методы, которые наследуются другими классами игровых объектов.

**Поля**:

* posX (положение по X): Целочисленное поле, обозначающее горизонтальную позицию объекта в комнате.
* posY (положение по Y): Целочисленное поле, обозначающее вертикальную позицию объекта в комнате.
* health (здоровье): Целочисленное поле, обозначающее количество здоровья объекта.

**Методы**:

* GameObject(int x, int y, int hp): Конструктор класса, который принимает три аргумента: координаты x и y, и количество здоровья hp, и инициализирует соответствующие поля объекта.
* virtual ~GameObject(): Виртуальный деструктор класса.
* virtual void update() = 0: Чисто виртуальная функция для обновления объекта.
* int getX() const: Метод, который возвращает горизонтальную позицию объекта.
* int getY() const: Метод, который возвращает вертикальную позицию объекта.
* void setX(int x): Метод, который устанавливает горизонтальную позицию объекта.
* void setY(int y): Метод, который устанавливает вертикальную позицию объекта.
* virtual void takeDamage(int damage): Виртуальный метод, который уменьшает количество здоровья объекта на величину damage.
* int getHealth() const: Метод, который возвращает текущее количество здоровья объекта.

## **Класс Character**

**Назначение**: Класс Character предназначен для представления игрового персонажа, управляемого игроком. Он наследуется от класса GameObject и добавляет функциональность, связанную с атакой и получением урона.

**Поля**:

* attackDamage (урон атаки): Целочисленное поле, обозначающее величину урона, наносимого при атаке.

**Методы**:

* Character(int x, int y): Конструктор класса, который принимает два аргумента: координаты x и y, и инициализирует соответствующие поля объекта, а также задает стандартное значение здоровья (100) и урона атаки (10).
* void update() override: Переопределенный метод для обновления состояния персонажа.
* void attack(GameObject\* target): Метод, который выполняет атаку на указанный объект target, нанося ему урон.
* void receiveDamage(int damage): Метод, который уменьшает количество здоровья персонажа на величину damage и выводит сообщение о получении урона.
* bool checkCollision(int x, int y) const: Метод, который проверяет, находится ли персонаж на указанных координатах x и y.

## **Класс Enemy**

**Назначение**: Класс Enemy предназначен для представления врагов в игре. Он наследуется от класса GameObject и добавляет функциональность, связанную с атакой и случайным передвижением.

**Поля**:

* attackDamage (урон атаки): Целочисленное поле, обозначающее величину урона, наносимого при атаке.

**Методы**:

* Enemy(int x, int y): Конструктор класса, который принимает два аргумента: координаты x и y, и инициализирует соответствующие поля объекта, а также задает стандартное значение здоровья (50) и урона атаки (5).
* void update() override: Переопределенный метод для обновления состояния врага, который включает случайное передвижение.
* void move(): Метод, который выполняет случайное передвижение врага в пределах комнаты.
* void takeDamage(int damage) override: Переопределенный метод, который уменьшает количество здоровья врага на величину damage и выводит сообщение о его смерти, если здоровье меньше или равно нулю.
* void attack(Character\* target): Метод, который выполняет атаку на указанный объект target, нанося ему урон.
* bool isDead() const: Метод, который проверяет, жив ли враг (имеет ли он здоровье больше нуля).

## **Класс Boss**

**Назначение**: Класс Boss предназначен для представления босса в игре. Он наследуется от класса Enemy и добавляет уникальные характеристики и функциональность.

**Поля**:

* bossName (имя босса): Строковое поле, хранящее имя босса.

**Методы**:

* Boss(int x, int y, const string& name): Конструктор класса, который принимает три аргумента: координаты x и y, и имя босса name, и инициализирует соответствующие поля объекта, а также задает увеличенное значение здоровья (200).
* void update() override: Переопределенный метод для обновления состояния босса.
* string getName() const: Метод, который возвращает имя босса.
* void takeDamage(int damage) override: Переопределенный метод, который уменьшает количество здоровья босса на величину damage и выводит сообщение о его смерти, если здоровье меньше или равно нулю.

## **Класс Item**

**Назначение**: Класс Item предназначен для представления предметов в игре. Он наследуется от класса GameObject и добавляет уникальные характеристики и функциональность.

**Поля**:

* name (название): Строковое поле, хранящее название предмета.

**Методы**:

* Item(int x, int y, const string& itemName): Конструктор класса, который принимает три аргумента: координаты x и y, и название предмета itemName, и инициализирует соответствующие поля объекта.
* void update() override: Переопределенный метод для обновления состояния предмета.
* string getName() const: Метод, который возвращает название предмета.

## **Класс Room**

**Назначение**: Класс Room предназначен для представления игровой комнаты, содержащей игровые объекты.

**Поля**:

* vector<GameObject\*> roomObjects (объекты комнаты): Вектор указателей на игровые объекты, находящиеся в комнате.

**Методы**:

* void addGameObject(GameObject\* obj): Метод, который добавляет игровой объект в комнату.
* void removeGameObject(GameObject\* obj): Метод, который удаляет игровой объект из комнаты.
* void updateRoom(): Метод, который обновляет состояние всех объектов в комнате.
* vector<GameObject\*>& getGameObjects(): Метод, который возвращает вектор указателей на игровые объекты в комнате.

## **Класс Inventory**

**Назначение**: Класс Inventory предназначен для представления инвентаря игрока, содержащего различные предметы.

**Поля**:

* vector<T\*> items (предметы): Вектор указателей на предметы, хранящиеся в инвентаре.

**Методы**:

* void addItem(T\* item): Метод, который добавляет предмет в инвентарь.
* void removeItem(T\* item): Метод, который удаляет предмет из инвентаря.
* T\* getItem(int index): Метод, который возвращает предмет по указанному индексу.
* int getSize() const: Метод, который возвращает количество предметов в инвентаре.

## **Класс GameSession**

**Назначение**: Класс GameSession предназначен для управления игровой сессией, включая инициализацию, обновление и отображение игрового процесса.

**Поля**:

* Room gameRoom (игровая комната): Объект класса Room, представляющий игровую комнату.
* Character\* player (игрок): Указатель на объект класса Character, представляющий персонажа игрока.
* Boss\* finalBoss (финальный босс): Указатель на объект класса Boss, представляющий финального босса.
* Inventory<Item> playerInventory (инвентарь игрока): Объект класса Inventory, представляющий инвентарь игрока.
* bool gameRunning (игра запущена): Булево поле, обозначающее состояние игры (запущена или нет).

**Методы**:

* GameSession(): Конструктор класса, инициализирующий начальное состояние игровой сессии.
* void initialize(): Метод, который инициализирует игровую комнату и размещает в ней игровые объекты.
* void update(): Метод, который обновляет состояние игровой комнаты и объектов.
* void drawGame(): Метод, который отображает игровое поле в текстовом формате.
* void showInventory(): Метод, который отображает инвентарь игрока.
* void handleInput(): Метод, который обрабатывает пользовательский ввод для управления персонажем.
* bool isGameRunning() const: Метод, который возвращает состояние игры (запущена или нет).
* void endGame(): Метод, который завершает игровую сессию.
* `void check

# **Диаграмма классов**

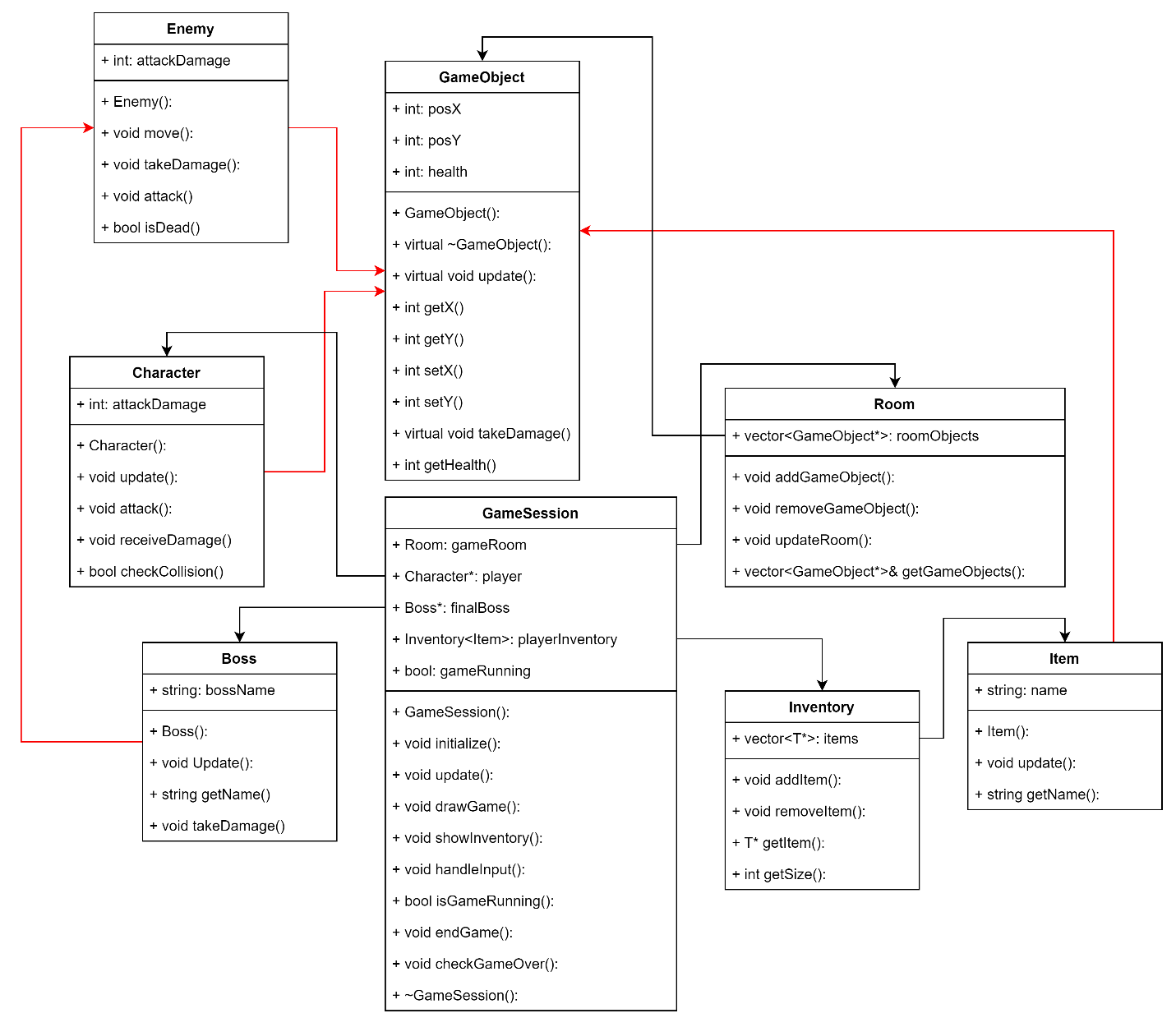


Figure 1 Диаграмма классов

# **Листинг программы**

#include <iostream>  
#include <vector>  
#include <cstdlib>  
#include <ctime>  
#include <stdexcept>  
#include <string>  
#include <iomanip>  
#include <algorithm>  
#include <chrono>  
#include <thread>  
#include <conio.h>  
  
using namespace std;  
  
*// Размеры игровой комнаты*const int ROOM\_WIDTH = 10;  
const int ROOM\_HEIGHT = 10;  
  
*// Символы для отображения элементов*const char PLAYER\_CHAR = '@';  
const char ENEMY\_CHAR = 'E';  
const char BOSS\_CHAR = 'B';  
const char ITEM\_CHAR = 'I';  
const char WALL\_CHAR = '#';  
const char EMPTY\_CHAR = '.';  
  
*// Цветовые коды ANSI для текста*const string PLAYER\_COLOR = "\033[1;32m"; *// Зеленый*const string ENEMY\_COLOR = "\033[1;31m"; *// Красный*const string BOSS\_COLOR = "\033[1;35m"; *// Магента*const string ITEM\_COLOR = "\033[1;33m"; *// Желтый*const string WALL\_COLOR = "\033[1;37m"; *// Белый*const string EMPTY\_COLOR = "\033[0m"; *// Сброс цвета  
  
// Цветовые коды ANSI для фона*const string BG\_COLOR = "\033[1;40m"; *// Черный фон  
  
  
// Базовый класс игровых объектов*class GameObject {  
protected:  
 int posX;  
 int posY;  
 int health;  
public:  
 GameObject(int x, int y, int hp) : posX(x), posY(y), health(hp) {}  
 virtual ~GameObject() {}  
  
 virtual void update() = 0; *// Виртуальная функция для обновления объекта  
  
 // Геттеры и сеттеры* int getX() const { return posX; }  
 int getY() const { return posY; }  
 void setX(int x) { posX = x; }  
 void setY(int y) { posY = y; }  
  
 virtual void takeDamage(int damage) {   
 health -= damage;  
 if (health <= 0) {  
 health = 0;  
 }  
 }  
  
 int getHealth() const { return health; }  
};  
  
class Character : public GameObject {  
protected:  
 int attackDamage;  
public:  
 Character(int x, int y) : GameObject(x, y, 100), attackDamage(10) {}  
  
 void update() override {  
 *// Логика обновления персонажа* }  
  
 void attack(GameObject\* target) {  
 target->takeDamage(attackDamage);  
 if (target->getHealth() <= 0) {  
 cout << "Target defeated!" << endl;  
 cout << "Press any key to continue..." << endl;  
 \_getch();  
 }  
 }  
  
 void receiveDamage(int damage) {  
 takeDamage(damage);  
 cout << "Player received " << damage << " damage!" << endl;  
 }  
  
 bool checkCollision(int x, int y) const {  
 return (posX == x && posY == y);  
 }  
};  
  
class Enemy : public GameObject {  
protected:  
 int attackDamage;  
public:  
 Enemy(int x, int y) : GameObject(x, y, 50), attackDamage(5) {}  
  
 void update() override {  
 move();  
  
 }  
  
 void move() {  
 int direction = rand() % 4;  
 int newX = posX, newY = posY;  
 switch (direction) {  
 case 0: newY--; break; *// вверх* case 1: newY++; break; *// вниз* case 2: newX--; break; *// влево* case 3: newX++; break; *// вправо* }  
  
 *// Проверка на границы комнаты и наличие стены* if (newX > 0 && newX < ROOM\_WIDTH - 1 && newY > 0 && newY < ROOM\_HEIGHT - 1) {  
 posX = newX;  
 posY = newY;  
 }  
  
 }  
  
 void takeDamage(int damage) override {  
 GameObject::takeDamage(damage);  
 if (getHealth() <= 0) {  
 cout << "Enemy died!" << endl;  
 }  
 }  
  
 void attack(Character\* target) {  
 target->receiveDamage(attackDamage);  
 }  
  
 bool isDead() const {  
 return getHealth() <= 0;  
 }  
  
};  
  
  
class Boss : public Enemy {  
private:  
 string bossName;  
public:  
 Boss(int x, int y, const string& name) : Enemy(x, y), bossName(name) {  
 health = 200; *// У босса больше здоровья* }  
  
 void update() override {  
 *// Логика движения и атаки босса* }  
  
 string getName() const { return bossName; }  
  
 void takeDamage(int damage) override {  
 Enemy::takeDamage(damage);  
 if (getHealth() <= 0) {  
 cout << "Boss died!" << endl;  
 }  
 }  
};  
  
*// Класс предмета*class Item : public GameObject {  
private:  
 string name;  
public:  
 Item(int x, int y, const string& itemName) : GameObject(x, y, 0), name(itemName) {}  
  
 void update() override {  
 *// Логика обновления предмета* }  
  
 string getName() const { return name; }  
};  
  
*// Класс комнаты*class Room {  
private:  
 vector<GameObject\*> roomObjects;  
public:  
 void addGameObject(GameObject\* obj) {  
 roomObjects.push\_back(obj);  
 }  
  
 void removeGameObject(GameObject\* obj) {  
 auto it = find(roomObjects.begin(), roomObjects.end(), obj);  
 if (it != roomObjects.end()) {  
 roomObjects.erase(it);  
 }  
 }  
  
 void updateRoom() {  
 for (auto obj : roomObjects) {  
 obj->update();  
 }  
 }  
  
 vector<GameObject\*>& getGameObjects() {  
 return roomObjects;  
 }  
};  
  
*// Класс инвентаря*template<typename T>  
class Inventory {  
private:  
 vector<T\*> items;  
public:  
 void addItem(T\* item) {  
 items.push\_back(item);  
 }  
  
 void removeItem(T\* item) {  
 auto it = find(items.begin(), items.end(), item);  
 if (it != items.end()) {  
 items.erase(it);  
 }  
 }  
  
 T\* getItem(int index) {  
 if (index >= 0 && index < items.size()) {  
 return items[index];  
 }  
 return nullptr;  
 }  
  
 int getSize() const {  
 return items.size();  
 }  
};  
  
*// Класс игровой сессии*class GameSession {  
private:  
 Room gameRoom;  
 Character\* player;  
 Boss\* finalBoss;  
 Inventory<Item> playerInventory;  
 bool gameRunning;  
 int usedMoves = 0;  
  
public:  
 GameSession() : player(nullptr), finalBoss(nullptr), gameRunning(true) {}  
  
 void initialize() {  
 *// Создание персонажа* player = new Character(5, 5);  
  
 *// Создание предметов и добавление их в комнату* Item\* healthPotion = new Item(2, 3, "Health Potion");  
 gameRoom.addGameObject(healthPotion);  
  
 *// Создание врагов и добавление их в комнату* for (int i = 0; i < 3; ++i) {  
 gameRoom.addGameObject(new Enemy(rand() % ROOM\_WIDTH, rand() % ROOM\_HEIGHT));  
 }  
  
 *// Создание босса* finalBoss = new Boss(8, 8, "Final Boss");  
  
 *// Добавление босса в комнату* gameRoom.addGameObject(finalBoss);  
  
 *// Добавление игрока в комнату* gameRoom.addGameObject(player);  
  
 }  
  
 void update() {  
 gameRoom.updateRoom();  
  
 usedMoves++;  
  
 *// Удаление убитых врагов* auto& objects = gameRoom.getGameObjects();  
 objects.erase(remove\_if(objects.begin(), objects.end(), [](GameObject\* obj) {  
 if (auto enemy = dynamic\_cast<Enemy\*>(obj)) {  
 return enemy->isDead();  
 }  
 return false;  
 }), objects.end());  
  
 checkGameOver(); *// Добавление проверки на завершение игры* }  
  
 void drawGame() {  
 *// Очистка экрана*#ifdef \_WIN32  
 system("cls");  
#else  
 system("clear");  
#endif  
  
 *// Отображение игрового поля* cout << "\n=== Carnage ===" << endl;  
 for (int y = 0; y < ROOM\_HEIGHT; ++y) {  
 for (int x = 0; x < ROOM\_WIDTH; ++x) {  
 bool objectFound = false;  
 for (auto obj : gameRoom.getGameObjects()) {  
 if (obj->getX() == x && obj->getY() == y) {  
 if (dynamic\_cast<Character\*>(obj)) {  
 cout << PLAYER\_COLOR << PLAYER\_CHAR << EMPTY\_COLOR << " ";  
 }  
 else if (dynamic\_cast<Enemy\*>(obj)) {  
 cout << ENEMY\_COLOR << ENEMY\_CHAR << EMPTY\_COLOR << " ";  
 }  
 else if (dynamic\_cast<Boss\*>(obj)) {  
 cout << BOSS\_COLOR << BOSS\_CHAR << EMPTY\_COLOR << " ";  
 }  
 else if (dynamic\_cast<Item\*>(obj)) {  
 cout << ITEM\_COLOR << ITEM\_CHAR << EMPTY\_COLOR << " ";  
 }  
 objectFound = true;  
 break;  
 }  
 }  
 if (!objectFound) {  
 *// Пустое место (пустая комната или стена)* cout << WALL\_COLOR << (x == 0 || x == ROOM\_WIDTH - 1 || y == 0 || y == ROOM\_HEIGHT - 1 ? WALL\_CHAR : EMPTY\_CHAR) << EMPTY\_COLOR << " ";  
 }  
 }  
 cout << endl;  
 }  
 cout << "===========================" << endl;  
 }  
  
 void showInventory() {  
 *// Очистка экрана*#ifdef \_WIN32  
 system("cls");  
#else  
 system("clear");  
#endif  
  
 *// Установка цвета фона* cout << BG\_COLOR;  
  
 *// Отображение инвентаря* cout << "\n=== Inventory ===" << endl;  
 for (int i = 0; i < playerInventory.getSize(); ++i) {  
 Item\* item = playerInventory.getItem(i);  
 if (item) {  
 cout << i + 1 << ". " << item->getName() << endl;  
 }  
 }  
 cout << "=================" << endl;  
  
 *// Ожидание нажатия клавиши для возврата к игре* cout << "Press any key to continue..." << endl;  
 \_getch();  
 }  
  
 void handleInput() {  
 if (\_kbhit()) {  
 char action = \_getch(); *// Использование \_getch() вместо getchar() для немедленного считывания клавиши* switch (action) {  
 case 'w': {  
 player->setY(max(1, player->getY() - 1));  
 break;  
 }  
 case 's': {  
 player->setY(min(ROOM\_HEIGHT - 2, player->getY() + 1));  
 break;  
 }  
 case 'a': {  
 player->setX(max(1, player->getX() - 1));  
 break;  
 }  
 case 'd': {  
 player->setX(min(ROOM\_WIDTH - 2, player->getX() + 1));  
 break;  
 }  
 case 'i': {  
 showInventory();  
 break;  
 }  
 case ' ': {  
 *// Проверка наличия врага в ближайших клетках* bool enemyFound = false;  
 for (auto obj : gameRoom.getGameObjects()) {  
 if (obj != player && abs(obj->getX() - player->getX()) <= 1 && abs(obj->getY() - player->getY()) <= 1) {  
 player->attack(obj);  
 enemyFound = true;  
 break;  
 }  
 }  
  
 *// Проверка на остаток врагов* bool anyEnemiesLeft = false;  
 for (auto obj : gameRoom.getGameObjects()) {  
 if (dynamic\_cast<Enemy\*>(obj)) {  
 anyEnemiesLeft = true;  
 break;  
 }  
 }  
  
 *// Если врагов не осталось, выводим сообщение и ожидаем нажатия клавиши, кроме движений* if (!anyEnemiesLeft && enemyFound) {  
 cout << "All enemies defeated! Press any key (except movement keys) to continue..." << endl;  
 char key;  
 do {  
 key = \_getch();  
 } while (key == 'w' || key == 'a' || key == 's' || key == 'd');  
 }  
 break;  
 }  
 }  
 }  
  
 *// Проверка наличия предмета в текущей позиции игрока* for (auto obj : gameRoom.getGameObjects()) {  
 if (auto item = dynamic\_cast<Item\*>(obj)) {  
 if (player->checkCollision(item->getX(), item->getY())) {  
 *// Подбор предмета* playerInventory.addItem(item);  
 gameRoom.removeGameObject(item);  
 cout << "Picked up " << item->getName() << "!" << endl;  
 *// Ожидание нажатия клавиши для возврата к игре* cout << "Press any key to continue..." << endl;  
 \_getch();  
 break;  
  
 }  
 }  
 }  
 }  
  
 bool isGameRunning() const {  
 return gameRunning;  
 }  
  
 void endGame() {  
 gameRunning = false;  
 }  
  
 void checkGameOver() {  
 bool anyEnemiesLeft = false;  
 for (auto obj : gameRoom.getGameObjects()) {  
 if (dynamic\_cast<Enemy\*>(obj)) {  
 anyEnemiesLeft = true;  
 break;  
 }  
 }  
 if (!anyEnemiesLeft) {  
 cout << "All enemies have been defeated. Game Over!" << endl;  
 cout << "It took you " << usedMoves << " frame refreshes." << endl;  
 \_getch();  
 endGame();  
 }  
 }  
  
 ~GameSession() {  
 delete player;  
 delete finalBoss;  
 for (auto obj : gameRoom.getGameObjects()) {  
 delete obj;  
 }  
  
 }  
};  
  
*// Функция для отображения главного меню*void displayMainMenu() {  
 cout << "=== Main Menu ===" << endl;  
 cout << "1. Start Game" << endl;  
 cout << "2. Exit Game" << endl;  
 cout << "=================" << endl;  
 cout << "Enter your choice: ";  
}  
  
int main() {  
 srand(static\_cast<unsigned>(time(0))); *// Инициализация генератора случайных чисел* bool exitGame = false;  
 while (!exitGame) {  
 *// Отображение главного меню* displayMainMenu();  
  
 *// Считывание выбора пользователя* char choice;  
 cin >> choice;  
  
 switch (choice) {  
 case '1': {  
 *// Игровая сессия* GameSession gameSession;  
 gameSession.initialize();  
  
 while (gameSession.isGameRunning()) {  
 gameSession.update();  
 gameSession.drawGame();  
 gameSession.handleInput();  
 this\_thread::sleep\_for(chrono::milliseconds(100));  
 }  
 break;  
 }  
 case '2': {  
 *// Выход из игры* cout << "Exiting Game..." << endl;  
 exitGame = true;  
 break;  
 }  
 default:  
 cout << "Invalid choice. Please enter 1 or 2." << endl;  
 break;  
 }  
 }  
}

# **Тесты**

В начале старта программы пользователя встречает меню с двумя опциями:

* 1. Начать игру;
  2. Выйти из игры.

Скрин

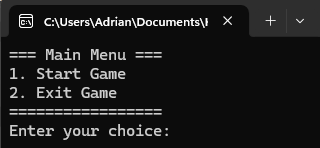


Figure 2 Игровое меню

При выборе второй опции программа закрывается.

При выборе первой опции стартует игра:

Пользователь играет за персонажа, значок которого помечается на экране как зеленый знак ‘@’. Персонаж появляется в комнате 10х10, так же с ним появляются в комнате 4 врага, один из которых является боссом. Противники помечаются красным знаком ‘E’ Враги двигаются в случайном направлении в комнате, кроме босса, который стоит в правом нижнем углу комнаты. В комнате так же появляется предмет, который помечается как желтый знак ‘I’.

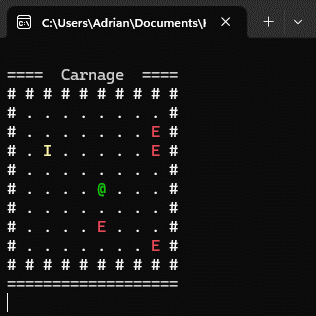


Figure 3 Игровое пространство

Игрок перемещается клавишами ‘W’, ‘A’, ‘S’, ‘D’, которые передвигают персонажа в направлениях «вверх», «влево», «вниз», «вправо» соответственно.

Персонаж может подобрать предмет, который пополняет его инвентарь на 1 предмет: зелье здоровья.

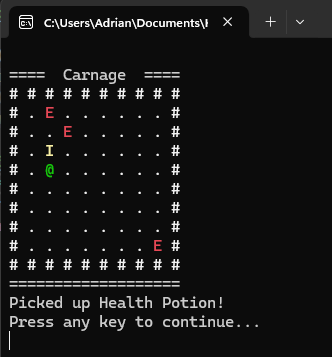


Figure 4 Поднятие предмета

Которое окажется позже в инвентаре.

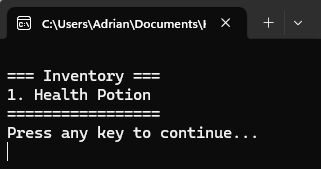


Figure 5 Меню инвентаря

Персонаж так же может атаковать противников. Для этого нужно находиться в соседней клетке или этой же клетке, где есть враг, и нажать кнопку «Space», для атаки. При достаточном количестве атак противник умирает.

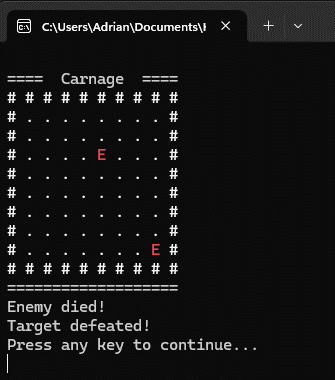


Figure 6 Убийство врага

При убийстве противника он пропадает с карты. Убив всех противников, игра завершается и выводится сообщение о том, сколько обновлений экранов заняла победа.

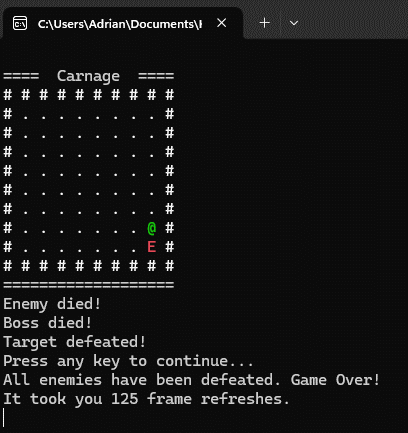


Figure 7 Убийство финального босса, конец игры и счет

**Заключение**

В процессе разработки текстовой игры "Carnage" я углубил свои знания и понимание объектно-ориентированного программирования. В частности, я освоил использование классов и наследования для эффективного организации игровых объектов, таких как персонажи, враги и предметы.

Важными аспектами стали понимание и использование виртуальных методов для реализации полиморфизма, что позволило различным типам объектов вести себя по-разному в игровой среде.

Опыт работы с контейнерами STL, такими как векторы для хранения игровых объектов и инвентаря игрока, показал мне важность эффективного управления данными в больших проектах. Использование указателей и динамическое выделение памяти помогли обеспечить гибкость и эффективность в управлении ресурсами.

Все эти аспекты в совокупности составили ценный опыт, который я смог применить в этом проекте, а также который будет полезен в будущем для создания и развития более сложных и интересных программных продуктов.

Github: <https://github.com/KopzhasharovAzamat/OOP_term>