Robert Mays

Rogue Descent: The Last Hack

A circuit board

Description automatically generated

# Executive Summary

This rogue-like game’s design process addresses each component of the necessary code in a manner that is consistent with achieving increasingly complex testing capabilities during development.

The game pits a lone character against hordes of foes, demonic and otherwise, as they delve further into the earth. The levels are a combination of static levels and levels generated with procedural methods. These levels can contain monsters, items, and special random events, and any number of additional possibilities in time with updates. As the character explores deeper levels, the monsters that can spawn will gain in strength and new items will become available to the characters.

The code is designed from a standpoint that appreciates the benefits of planning for modular expansion of game complexity. Adding new character classes will become an option if time allows. As the process is completed over the course of the schedule, some functionality may be added beyond the scope of the project.

Upon project completion, the base game is completely playable.

# Beginning a Game

# 

# As the player begins the game, they are faced with some immediate choices. The player can choose between creating a new game, or loading a saved game. If the player creates a new character, they must choose a new name and a criminal profession for their character. If they load a saved game, the player will be initialized based on saved character data from a previous game. These choices will dictate how the player’s initial attributes are generated, and how they advance as the player grows in power.

# Exploring the Game World

# 

# As the game begins, the Level 1 map loads, is processed, and the player may begin moving about the game world. The character attributes are displayed below the map, and refreshed every turn. The (w/a/s/d) keys provide movement in the cardinal directions and the player is represented by ‘@’. Floor areas are displayed as ‘.’, stone walls as ‘#’, and enemies are represented by different letters. Stairs leading up are noted as ‘<’, while stairs leading down to a lower level are represented by ‘>’.

# Interacting with the Game World

# 

# The game world is dangerous for players, and they will encounter many foes and challenges as they play the game. Here, we see the player (Ralph the Smuggler) attempting to fight off another inmate, as he breaks out of his cell and searches for a way out. Enemy AI will track the players movements and give chase if the player gets too close, running them down and attacking players if they get close enough. If the player moves into an enemy’s space, it will initiate a player attack against the enemy, pitting the players’ damage and damage type against the varied resistances and armor of the target enemy. If the enemy dies, the player gains experience and the enemy is removed from the game. If the player dies, the game is over.

# Gaining Power

# 

# The player can increase his power by slaying his foes, which causes the player to increase in level as they progress through the game. The attribute increases that occur when a level is gained are added depending on the character’s chosen criminal profession. Here, we can see that Ralph’s abilities have increased after vanquishing enough foes by looking at the player attribute bar below the map.

# 

# Saving and Loading Player Data

By pressing the ‘g’ key, the player can save the game at any time. This data is saved in a text file, separated by player attribute, that is later retrieved when loading the game.

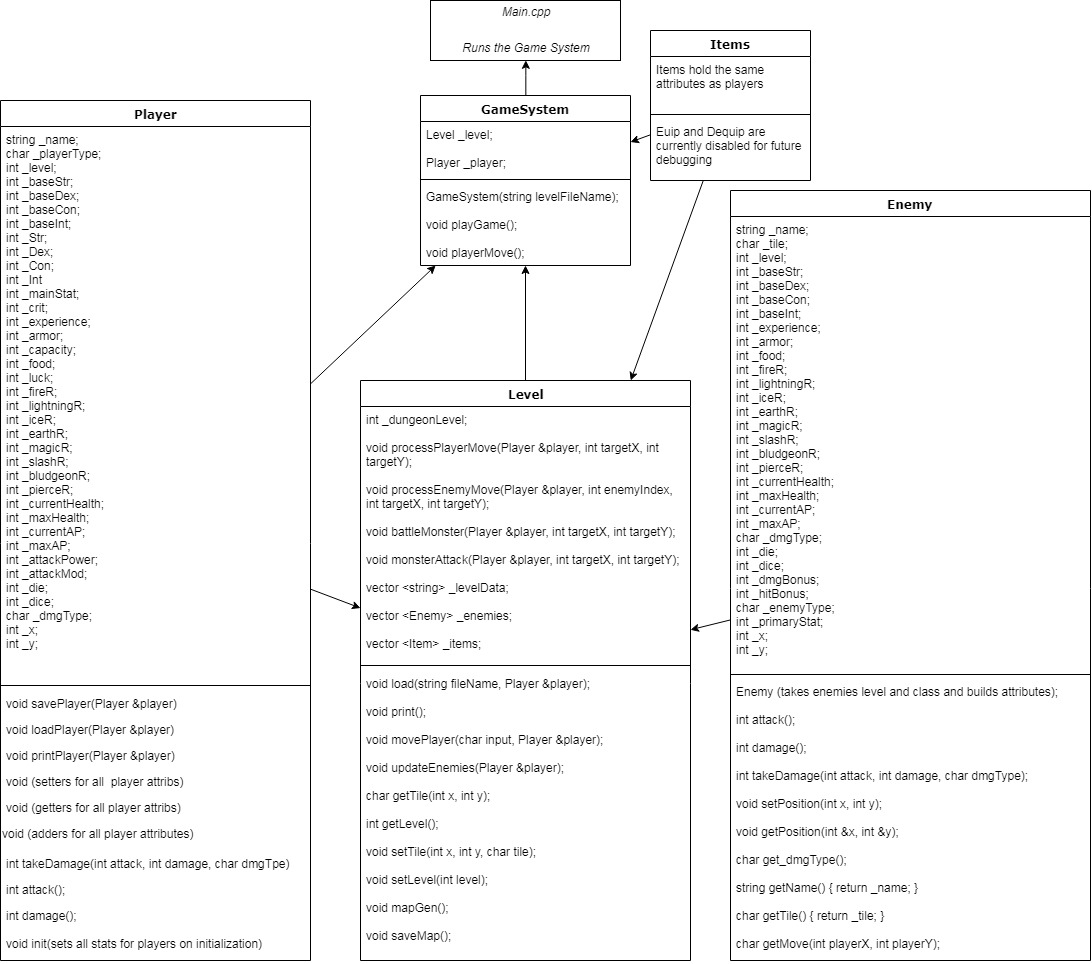
# 





The player is reinitialized using the saved player data from the last saved game. This functionality allows a player to enjoy the game at their leisure, without a worry for time constraints.

# System and Code Architecture



This is the map of the system’s structure, with data from the classes fed into the level, and then the game system, which is then operated by the main program. Many class functions are available in each class, and they all work together to create the level and run the game system.

## 

## Source Code Structure

The following is a summary of the source code directories and their contents:

|  |  |
| --- | --- |
| **Code Directory** | |
| **Directory** | **Usage** |
| RDLH\_Final | This directory contains all of the .cpp, header, and text files (map.txt, saveMap.txt, and Level1.txt), and the main executable file that runs the game. All source code resides within this directory. The text file for the first level must be present in the active directory when running the .exe file |
|  | |

# Executables

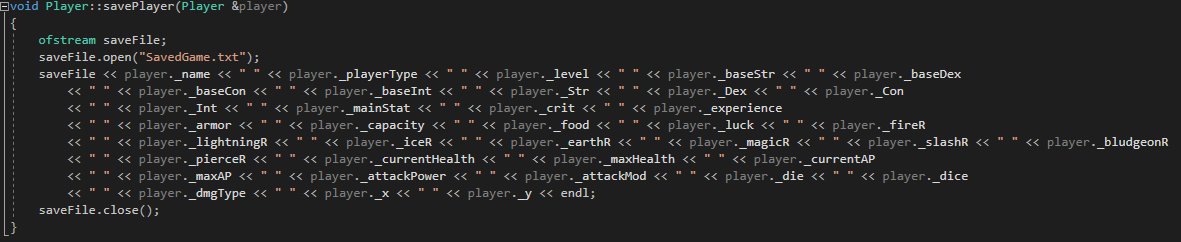
### RDLH\_Final (RDLH\_Final.exe)

This executable file runs the game, and requires the starting map file “Level1.txt” to load.

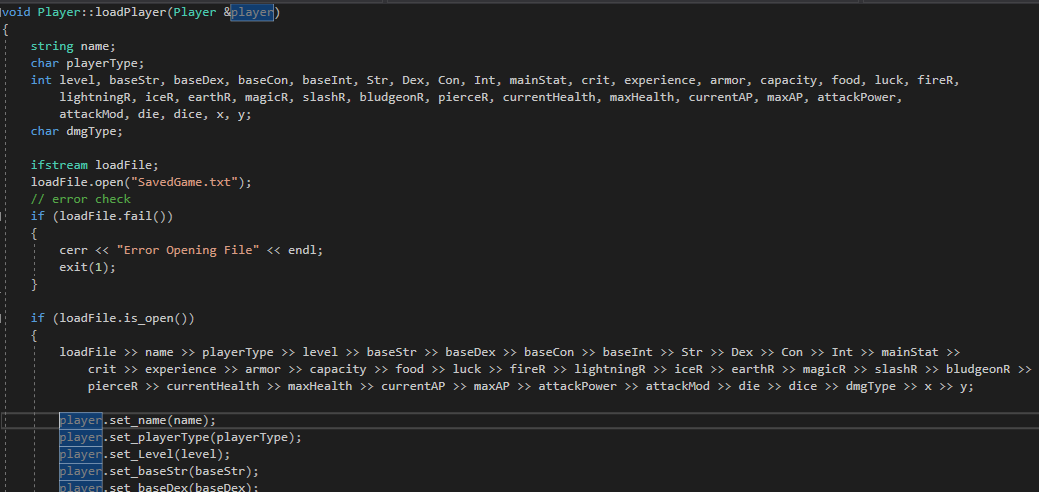
# Code Architecture

## This software has been organized to separate functionality of the different classes such as items, players, enemies, levels, and the game system itself. The design is intentionally modular, to allow for future growth of the software.

## **Data Storage**

During the game, the player will likely want to save their progress. To accomplish this, the player’s attributes are copied into the saveGame.txt in the game directory. 

This code helps load that data back into the player’s attribute set when they load the saved game. Once the data is read, it is copied into the player’s private attribute variables.



Programming Language

This program was built using the C++ programming language using Visual Studio 2017. No special dependencies or plugins were used in the generation of the source code.

Project Classes

Classes within the project are used to abstract re-usable pieces of code. Classes are also used to group related values, known as properties. Rogue Descent : Last Hack utilizes these classes:

### **GameSystem | GameSystem.cpp**

This class contains the functionality needed to assemble the class data from the other classes and run the game.

GameSystem(string levelFileName): This initializes the game, allowing the player to start the game at the beginning, or load a saved game.

Void playGame(): This runs the game until the game is over, printing the level map and moving the character and enemies. This function also restored 1 HP to the players current health total after every turn to allow them to recover following a battle.

Void playerMove(): This function takes user keyboard input and then applies it towards player interaction with the game world.

### **Player| Player.cpp**

This class contains functionality needed to store player variables and manipulate them. Each attribute has a function for setting, getting, or adding/subtracting from the value being stored. In addition to attribute manipulation, this functionality also resides within the player class:

Void init(player attributes); This function takes inputs and applies them toward player generation and attribute setting. Internal algorithms are used to generate tertiary attributes using the primary attributes.

Int attack(); This function performs a player attack roll.

Int damage(): This function performs a player damage roll.

Void savePlayer(Player &player); This function stores player data in a text file for the saved game.

Void loadPlayer(Player &player); This function loads player data and performs a player initialization based on the data, loading the stored attributes into the current player.

Void printPlayer(Player &player); This function prints the player data on the screen, allowing for the player attribute information display below the game map.

Int takeDamage(int attack, int damage, char dmgType); This function determines how much damage a player, if any, takes during an attack from an enemy. This is determined by comparing the enemie’s attack attributes to the player’s defensive attributes.

### **Enemy | Enemy.cpp**

This class is somewhat similar to the player class, but with some changes. The enemy is assigned a character called “tile”, which is the letter of the alphabet displayed on the map that represents the enemy, instead of an ‘@’ symbol. The enemy shares similar functionality for the attack(), damage(), and takeDamage() functions. In addition to those and setters/getters, there are:

Enemy(enemy attributes) : The enemy class’ initializing function determines the power level of the enemy based on the enemy’s type and level.

Char getMove(int playerX, int PlayerY): This establishes the AI of the enemies, allowing them to move and hunt down players on the (x,y) axis and attack players if they come to close enough for the enemies to notice them.

### **Items | Items.cpp**

The Item class serves to allow a character to equip an item to add its attributes to their own, and to remove an item which reverts the characters’ attributes to normal . As such, items hold the same variables as players.

### **Level | Level.cpp**

The Level class has a wide variety of responsibilities revolving around map generation, player and enemy movement, display generation, map saving, combat functionality, and vectors to hold level data, enemies, and items. The level class keeps track of the game’s difficulty level as well.

Void load(string fileName, Player &player): The load function generates a level based on the contents of a text file. It then reads the ASCII text file, and processes the map into level data. The level data is then parsed to determine if there are letters or other symbols that need to be initialized as enemies or items and allocated attributes to them based on the letter or symbol. For example:

case 's': // skeleton

\_enemies.push\_back(Enemy("skeleton", tile, 'd', 'B', 1, 0, 0, 0, 0, 0, 0, 0, 0, 50));

\_enemies.back().setPosition(j, i);

This code adds a skeleton enemy to the enemy vector at the positions of the letters ‘s’ on the map, while this code below adds a sword item to the item vector at the positions of the symbol ‘/’ on the map.

case '/': // sword

\_items.push\_back(Item("long sword", tile, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 1, 'S'));

\_items.back().setPosition(j, i);

Void print(): This function prints the level residing in level data onto the terminal.

Void movePlayer(char input, Player &player): This function processes the movement input from the Game System (w/a/s/d) and repositions the player and the ‘@’ symbol on the map accordingly. If the player typed ‘g’, the player and map will be saved.

Void updateEnemies(Player &player): Enemies are moved in cardinal directions by this function based on the relative location to the player to them.

Char getTile( int x, int y): This returns the character on the coordinates entered. There is a set function for this as well.

Int getLevel(); This function returns the dungeon level, which is used to track the difficulty level of the dungeon maps generated based on the level of the player/character. There is a set function to match.

Void mapGen(): The mapGen function creates a map with random floor and wall distributions, and places the character, a stairway leading up, and a stairway leading down on the map. This function also randomly places enemies on the map based on the depth, or number of floors below the first level the player is. Then, the map is written to map.txt.

Void saveMap(): This writes the current map.txt to saveMap.txt, which is used to restore the saved games of players.

Void processPlayerMove(Player &player, int targetX, int targetY): This function is used to determine what happens when a player tries to move into a particular (x,y) coordinate. If that tile is ‘#’, nothing happens because the player is bumping into a wall. If that tile is ‘.’, it swaps location with the player and the @ symbol on the map. Other functionality resides here for going up and down stairs, which calls the mapGen function and then loads a new map.

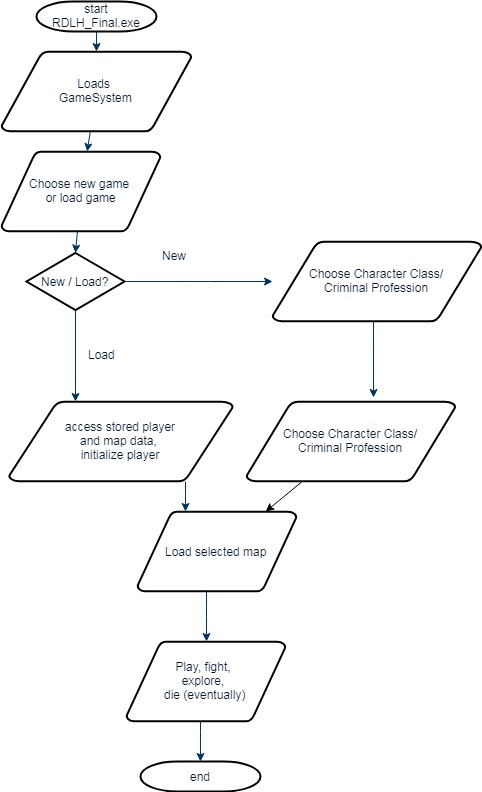
Void processEnemyMove(Player &player, int enemyIndex, int targetX, int targetY): This function processes enemy movement in a manner not unlike the player’s function.

Void battleMonster(Player &player, int targetX, int targetY): Combat between a player and an adversary is performed by calling this function. The attack and damage rolls of the player are accessed and processed to generate a simulated attack based on the player’s attributes. If the target is successfully hit, damage is applied to that enemy, and if that enemy dies, the player receives experience points based on the monster’s level. which invokes the enemy Int takeDamage(int attack, int damage, char dmgType) funtion which returns the experience value of the enemy if the enemy dies, which then calls the player add\_Exp(int experience, char playerType) function to add that amount of experience to the character and increase their attributes differently depending on their criminal profession.

Void monsterAttack(Player &player, int targetX, int target Y): In order for combat to work, there must be a function that executes a monster attack if a monster tries to move into the player’s square. The enemy’s attack is calculated based on its level, type and attributes, and if the player’s HP total is reduced to 0, a grisly message is displayed and the game ends.

Program Start and End Flow

The program flow begins with a series of player generation choices. Then, once the player and map have been generated or loaded from the saved game, the player can move about the map. At this point the player battles enemies explores to gain powers relative to their character class or criminal profession, until they are eventually vanquished. The challenge lies in seeing how many floors below the surface the player can descend to before that happens. This is a diagram explaining the general flow of the program:



Summary

The player, an escaping inmate, must fight there way out of prison and into a dungeon filled with horrors below. There they face varied enemies and learn to slay or avoid them in order to advance further into the deep. The only thing standing between escape and defeat are the choices they make.

Rogue Descent: The Last Hack is a system is designed to create a playable game environment, save and load data, and leave room for further expansion and updates. The system includes classes for the Player, for Enemies, Levels, Items, and the Game System. These classes and their functions are integrated into the game system to create an organized and dynamic gameplay experience depending on both random luck and player skills.

# APPENDIX B (BUILD AND RELEASE PROCESS)

This build is a basic version of the game system that will be improved as time goes on. Periodically, the GitHub repository for this system will be updated. To update to a newer version of the game, simply download the most recently uploaded version of the RDLH\_Final.exe.

# APPENDIX C (CLIENT INSTALLATION INSTRUCTIONS)

To run the game, make sure you download the Level1.txt text file and place it in the same folder as RDLH\_Final.exe. Players have the option to modify their map text files manually to create a custom game experience. If a player decides to do this, it is recommended to save a copy of Level1.txt elsewhere as a backup.

# APPENDIX D (DEVELOPER SETUP INSTRUCTIONS)

This code was generated using Microsoft Visual Studio 2017. To modify the source files, simply copy the project folder into the project directory of a machine running the appropriate version of Visual Studio. No additional requirements exist for development or debugging.