# Dungeon Game Report

Gerard Knight

Contents

[Dungeon Game Report 1](#_Toc196676522)

[Specification 3](#_Toc196676523)

[Combat 4](#_Toc196676524)

[Enemies 6](#_Toc196676525)

[Player Stats 7](#_Toc196676526)

[Potions 8](#_Toc196676527)

[Floors and Rooms 9](#_Toc196676528)

[Settings 11](#_Toc196676529)

[Misc Functions 12](#_Toc196676530)

[main.cpp 13](#_Toc196676531)

[Notes on the programming 14](#_Toc196676532)

[Use case diagram 15](#_Toc196676533)

[Play testing 16](#_Toc196676534)

[Evaluation 23](#_Toc196676535)

## Specification

The goal of this project is to create a text-based dungeon-crawler game. The main elements of this are combat, exploration and item management.

Combat is to be handled in a turn-based fashion, where the player and enemy will take turns making different moves. Exploration involves moving from room to room and dealing with any combat encounters and items in the room, as well as deciding where to move next. I intend for the exact layout of these rooms to be randomly generated, and maze like. Item management is to be simple, with the player expected to find potions with different effects.

The player will lose if their health reaches 0. They will win once they enter a treasure room. I will try to include a boss in this room, with more complex and difficult mechanics. Should the game prove too difficult to complete in a first attempt, I will include options that make it easier, for the convenience of examiners.

In terms of technical specifications, the main goal is scalability. It should be trivial to create new classes for new enemies, loot and rooms.

The rest of this report describes these mechanics in greater detail, and focuses only on what is actually implemented.

## Combat

Upon entering a room, an enemy may appear, initiating combat. The enemy will take one of several actions, and the player will be asked how they would like to respond. A table of moves can be seen below, with the enemy’s moves along the top, and the player’s response along the side.

* Draw: both sides hit each other.
* Damage: The enemy hits the player.
* Hit: The player hits the enemy.
* Feint-damage: The enemy hits the player.
* Repost: The enemy hits the player.
* Parry: Neither side hits the other. This may create an opening for the player, depending on the enemy.
* Nothing: Neither side hits the other.

Feint damage and reposts are functionally the same, but have different accompanying messages. A feint appears to be either a thrust or a slash, but the player must react differently to avoid damage. A repost occurs only when the player does something foolish, such as attacking while the enemy is in a defensive stance.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Thrust** | **Slash** | **Feint** | **Guard** | **Rest** |
| **Thrust** | Draw | Draw | Damage | Repost | Hit |
| **Slash** | Damage | Damage | Draw | Repost | Hit |
| **Parry** | Parry | Parry | Damage | Nothing | Nothing |
| **Guard** | Damage | Nothing | Nothing | Nothing | Nothing |

This table is in the code as a 2D array of ints. The array contains an extra column at the beginning that is never accessed, and a column each for a thrust feint and a slash feint.

Enemy moves are not completely random, instead being decided according to a ‘combo’. Each enemy has multiple ‘combos’ which are selected from randomly. For most enemies, this is represented by a string, meaning they will always do the same thing in the same order. They are done such that, if the player is able to recognise the patterns, they will be able to avoid taking any damage. The enemy will always rest at the end of the combo, in order to give the player a reliable opening.

Combat will end when either the player or the enemy run out of health.

The goal of this system is to create something skill-based, so that in theory a skilled player is able to go through the whole game without taking any damage.

Combat is implemented using a class called Combat. The class consists entirely of static functions. Because there are multiple enemies, they are handled by passing in references to those enemies as arguments. This was the easiest way to deal with multiple enemies. Because there are multiple types of enemies, the functions make use of templates. There is only one player, so the player’s stats are taken directly.

While implementing this system, the strange behaviour of cin became apparent. If cin asks for an int or char and the player does not give one of these, the code will break. Therefore, player inputs will always be in the form of a string.

A screen shot of a computer program

AI-generated content may be incorrect.

## Enemies

The default class of Enemy has multiple protected variables, e.g. health. Most of these are fairly self-explanatory, as are most of the functions. The mechanic that allows you to stun enemies is a little more complex, as in earlier versions, hitting an enemy once allowed you to stun-lock them until they died. Some action was needed to prevent this, adding complication. Enemies may or may not be stunned when being parried or hit.

There are 4 different types of enemy used in the game: Small Goblins, Large Goblins, Knight and Golem, in ascending order of toughness. More difficult enemies are harder to stun, have more health and more complex combos. The latter two are used as bosses to layers, while the Large Goblin is used as the boss to the first layer and the common enemy of the last. The Enemy class is never directly used. Different damage levels for different enemies were not implemented, but their damage is affected by the difficulty.

A screenshot of a computer

AI-generated content may be incorrect.

## Player Stats

This class contains all the information about the player, as well as all the functions used to affect it. It includes things like the players health and damage output. The only function that is at all complicated is the floor reward function. This provides the player with bonuses depending on which floor they have completed. It was implemented here as it was simply more convenient than doing it from the Floor class.

The effects of potions found in the dungeon are also partially implemented here, since the potions do not have access to the variables needed to do it themselves.

A screen shot of a computer program

AI-generated content may be incorrect.

## Potions

There are 3 types of potion, as well as a default potion that is not found in the game. The player can find them randomly throughout the dungeon. The Invincibility potion is the strongest and only spawns on easy difficulty. The health potion refills the players health to maximum and the double damage potion gives the player doble damage for the next fight.

They work when calling the effect() function. This is implemented as a virtual function in the default case, and overridden in the children classes. This function is only called by the apply\_effect() function, which will only do so if they have any potions.

There is a function in player stats which clears any potion effects once the player leaves combat. This is needed as the temporary boost from the invincibility and double damage potions are only intended to last for one fight.

They, like the player stats, are implemented as singletons. Picking one up does not create another instance of the potion, but rather increments the “num\_carried” variable.

A screen shot of a computer code

AI-generated content may be incorrect.

## Floors and Rooms

The Floor class is the central class around which all the others are organised, essentially a main class. It is therefore by far the most complex in the game. This is largely due to the large number of menus, and the large size of these menus. They are quite complex to implement. They are state machines, with the state changing based on where in the menu the player is. The largest menu is the calm() function, which is over 140 lines and has 5 separate states.

The floor is responsible for storing Room objects in a 2d array. The room contains data about its location, which doors are open and descriptions of what is in the room. Different types of room were *not* implemented using inheritance, but instead rely on the room describing its contents. Needing to use different classes for rooms that have enemies, items etc. means that a room cannot have both. It also means that the type of room can’t change. This creates a problem, since the player is able to re-enter rooms they have left, so enemies need to be deleted once they are killed.

Enemies and items are implemented using a bool value for if it has them, and a string containing which type of item/enemy. E.g. a room may have an item type of “Health Potion”. By default the type is “none”. There are getter functions for all such values, and functions to remove them once the player takes the item or kills the enemy.

The initialisation process is quite strange. It uses a function separate to the constructor to change the values mentioned above. It cannot be called more than once. This was done to allow the initialisation with a) multiple, often random, arguments b) within a 2d array. The incredible awkwardness of doing this with a regular constructor meant that this was easier.

Doors can be open or closed depending on the location of the room in the floor. This is done to prevent the player leaving the bounds of the floor. The Room class is responsible for working out which doors to open and close. Each floor has a different size (5x5, 6x6 and 7x7). The 2d array is deliberately made too large to facilitate this.

The Floor first generates the locations of items and enemies, and stores these to 2d arrays of strings. The top and bottom rows are devoid of items and regular enemies. They are randomly placed in the intervening rows. The bottom row is reserved for the player’s entrance, while the top row has the trapdoor to the next floor. This room also contains the boss of that floor. The Floor then initialises the rooms, by passing in the string that tells them what type of item/enemy it has. Most will be “none”.

The Floor is responsible for keeping track of the player’s location within it, what they do in the rooms, when they leave the floor through the trapdoor and when they have the key to the trapdoor. The only function called from outside the class is the floor\_main() function. This function is an int, so it can tell the main function if the player has left the room because they have died or gone through the trapdoor. It calls the enter\_room() function in a while loop, which exits when the player leaves the floor. If the room has an enemy, it will then instantiate the type of enemy and immediately call the combat function. It then clears any potion effects and calls the calm() function. This takes the player to a menu, where they can check their stats, check their inventory of potions or move into the next room. If they are in the trapdoor room, they may travel through it.

A screen shot of a computer program

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

## Settings

This allows you to change the difficulty of the game, and access cheats. Difficulty affects player health, whether the player heals upon defeating an enemy, enemy damage and whether or not invincibility potions spawn. The difficulty options are “easy”, “medium” or “hard”. Difficulty is changed in an options menu at the beginning of the game, implemented in main.cpp. There, there is a hidden option to access cheats. The player can give themselves extra health, double damage or make themselves take no damage. Once the player starts the game proper, they cannot change them.

This class is also responsible for maintaining information that may be useful, such as how much health the player starts with.

If the assessor finds the game too challenging, type in “cheats” when in the options menu. This should not be necessary.

A screen shot of a computer program

AI-generated content may be incorrect.

## Misc Functions

This class contains utility functions. None of these functions have any side effects; they simply take arguments and return values. For instance, there is a function to tell if there is a particular element in an array, max and min functions and others. None of them are very complex.

A computer screen shot of a code

AI-generated content may be incorrect.

## main.cpp

This file contains the aforementioned opening menu. It is also responsible for instantiating Settings, Player\_stats and the various potions. Finally, the main function calls the floor\_main functions and the floor\_reward functions. It also prints the final lines. Floors are actually instantiated in a separate function, so that they are deleted once the program leaves the scope of that function.

## Notes on the programming

I learned much while doing this project.

To begin, this was my first time using state machines to create menus, rather than nested loops, which is much less tidy. While they still took up lots of room, I am generally happy with their implementation.

I am unsure if the initialisation of rooms was done in the best way. However I was unable to find a method online of doing so in a way which met my requirements, so I was forced to improvise.

My understanding is that when making header files, the .h files should be the ones included in the main file. However, this would not work, so I had to include the .cpp files. This project was coded using CLion. I am unsure if this is something done by CLion, or if it part of C++.

## Use case diagram

A diagram of a computer

AI-generated content may be incorrect.

Includes:

Extends:

Note: This only shows when a class is a variable inside another class. All classes reference the same player\_stats object. I have also neglected to include combat and misc functions, since these consist entirely of static functions.

## Evaluation

I would like to have implemented a more elegant solution than making an entirely new state machine for menus. This is probably the biggest weakness my implementation has. The implementation of cheats is a little fiddly, but there it should be easy enough to add new ones, e.g. starting with the key. The final boss is a little easy, the boss to the second floor is the hardest. Changing this is trivial though, and it’s not an issue with the implementation. The Floor function is more complex than I would have liked, though I’m unsure of how to solve this, except move this complexity to a new function, which seems a little silly.

Menus requiring cin to use strings was initially a problem. Without this, the code could either fail or go into an infinite loop if the player accidentally typed in the wrong thing. This cannot happen now. Another issue was the initialisation of Rooms. As stated previously, normal constructors were no good for this. My solution to this, while inelegant, worked well and is unlikely to cause issues.

It is my belief that the code uses the specifications laid out, or exceeds them. The gameplay is more complex and engaging than in the specification. It also implements some of the additional features recommended, such as items and new enemy types. The only exception to this was not using inheritance for different types of room, but I believe this was the correct choice for this project. Having different room classes just adds difficulty and complexity.