Code Design Analysis

Our code consists of a class named `gameObject` with the following attributes: x-position, y-position, the character displayed on the map, and a copy of the map. It has several methods, including one for initializing the object on the map, a method for printing the character, getters for the positions, and a virtual destructor.

We use this class to create objects that populate our map. We create three new classes that inherit the elements mentioned above from `gameObject`.

- **Class `Player`:** This class includes an additional method for movement on the map based on arrow keys and a boolean check to validate movement, ensuring the player does not move into walls or out of bounds. (Developed by Babis Poteridis)
- **Class `Prize`:** This is the class for our diamond prize. It contains a movement method and a method to reset the object's previous position on the map to '.' after moving.
- **Class `AutoPlayer`:** Similar to the `Player` class, this class includes the boolean check along with its own movement method. It uses the smart algorithm BFS (Breadth-First Search), which will be further analyzed. (Developed by Loukas Malfou)

To complete the design, we implemented a class for the map called `Labyrinth`. This class reads the `txt` file we created for our map and stores it in a vector, enabling us to later read each character with coordinates. It also includes a method to locate empty positions on the map.

Finally, to combine everything, we create the `gameEngine` object to manage the game's design. It includes an object from each class as an attribute and has four methods: one for initializing ncurses and setting the initial positions of objects, one for checking if the game has ended, one for updating object positions, and finally, a game loop method to keep the game running until the goal is reached.

Technical Issues

The AI player's algorithm is the most challenging part of the code, so we will analyze the BFS algorithm in detail.

First, we create two vectors. The first holds the distances of cells from the player's position, and the second holds the previous positions of each cell during the algorithm. We also use a queue for BFS.

Initially, we check if the queue is empty. At each iteration, we check the first element in the queue to see if its coordinates match those of the prize. If it hasn't reached the goal, it checks neighboring positions, and if a position is valid and hasn't been visited, the queue, distances, and previous positions are updated.

When the search is complete, we create a path from the player's position to the prize. This is done by reading the previous positions vector in reverse, creating the path. From this path, we identify the optimal next move and move the player to these coordinates.

Development Environment

The terminal used is Ubuntu on Windows, and the programs were developed in Visual Studio Code on Windows with ncurses.

- **Compile**: `g++ main.cpp GameObject.cpp Prize.cpp Player.cpp AutoPlayer.cpp Labyrinth.cpp GameEngine.cpp -o game -lncurses`
- **Run**: `./game map.txt`