دانشگاه بوعلی سینا

Rakab

Advanced programming final project first phase

Reza Namvaran 40212358042

Mahdi Sadeghi 40212358026

Dr. Sakhaei-nia

Spring 2024

Contents

Introduction:

This work report details the development of the first phase of advanced programming final project, a video game adaptation of the Persian version of the classic Condottiere board game, Rakab.

We implemented this video game using C++, focusing on object oriented programming principles.

All of the development process is available on our GitHub repository: https://github.com/Reza-namvaran/Rakab

Challenges:

1. Memory leakage: the main challenge that we have faced during this project was memory leakage. It caused a lot of run-time errors during the development process and took a lot of time from us, but we resolved this problem by using smart pointers (shared pointer) and tools like Valgrind, by automating the process of memory management.
2. Circular dependencies: Another significant challenge we encountered was circular dependencies. These dependencies led to several issues, including compilation problems and increased complexity in code maintenance. They often caused cascading changes when we had to modify one part of the code, leading to potential bugs and elongated debugging sessions.

Tools:

In this section we mention some of the useful tools that we used during the development process.

1. Git: We used Git as our version control system to manage changes in the project's source code effectively.

2. Github projects team planning: To manage and organize our tasks, we made use of GitHub Projects. This tool enabled us to create and assign tasks, set deadlines, and track progress through Kanban boards.

3. Valgrind: Valgrind was very useful for identifying and fixing memory-related issues in our source code. By running our application through Valgrind, we could detect memory leaks, invalid memory access, and other critical bugs, thus improving the stability and performance of our game. This tool was particularly useful for ensuring that our game ran smoothly across different systems without compromising on reliability.

4. CMake: We used CMake as our build system to manage the compilation process of our project.

Player Class:

This class represents a player in the game, and it contains all the necessary information about the player and its status in game

Such as score, captured lands, played cards, etc.

This class has composition relationship with class like Card, Land, PlayerSign.

Data members:

1. name (string) : the name of the player.
2. age (unsigned int) : the age of the player.
3. total_score (unsigned int) : total score of the player.
4. passed (bool) : this data member indicates that the player has passed its turn do others during a battle or not (which is initialized to false at the beginning).
5. hand (shared_ptr to Card) : the cards available in player's hand.
6. sign (shared_ptr to PlayerSign) : the sign that the player choose. Each player has a sign with a unique color.

This class has a parametrized constructor which gets name, age and color as its arguments, and then it initializes total score to zero, and creates an instance of player sign with the given color for the player.

Function member:

1. setters and getters: this class has setter and getter functions to assign or retrieve the value of each data member of this class, such as name, age, score, captured lands count, etc.
2. getCard(): this method is used for getting the cards that the player has in its hands or has been played. It gets an argument (bool hand) as a flag and if it was true it will return the cards that are in players hand, but if it was false it will return cards that have been played in the current battle (this argument is set to true by default)
3. addCard(): this method works as a setter for hand and played cards data members. It works by taking a vector of shared pointer to cards and a flag. If the flag was true, then the cards will be added to hand, and if not the will be added to played …
4. addLand(): if a player captures a new land this method will add that land to the captured lands.
5. clearPlayedCard(): after each battle this method will clear played cards.
6. playCard(): this method is used for playing with a specific card(it will remove the card from hand and adds it to the played cards).
7. resetPlayerData(): this method will refresh the data that need to be reinitialized for every battle, i.e. it will set the score to zero, passed to false, and it will clear played cards.

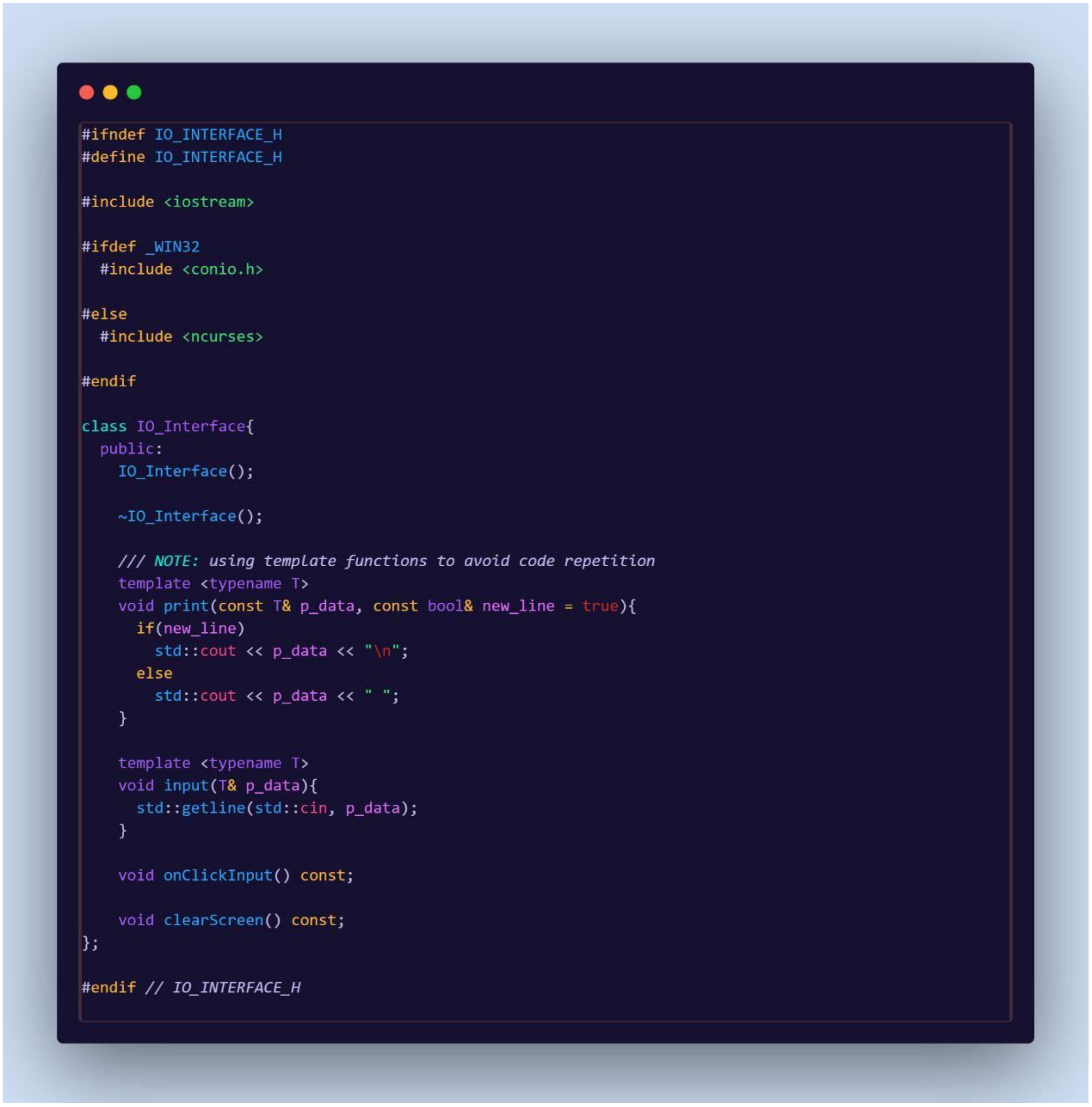
IO_Interface class:

This class works as an interface between console and the program, it can print given data from source code to the terminal or receive inputs from command line.

This class doesn't have any data members.

Function members:

1. print(): this is a template function that is used to print any type of data on the console screen.   It will receive the data and a flag that is used for determining if the data should end with the new line escape code(\n) or just a white space.
2. Input(): also a template function that is used for reading data from terminal input.
3. onClickInput(): this method is used for when we just need to enter a button to do something, e.g. pressing a button to exit a page or accept something.
4. clearScreen(): this method is used to clear console screen.


```
#ifndef IO_INTERFACE_H
#define IO_INTERFACE_H

#include <iostream>

#ifdef _WIN32
  #include <conio.h>

#else
  #include <ncurses>

#endif

class IO_Interface{
  public:
    IO_Interface();

    ~IO_Interface();

    /// NOTE: using template functions to avoid code repetition
    template <typename T>
    void print(const T& p_data, const bool& new_line = true){
      if(new_line)
        std::cout << p_data << "\n";
      else
        std::cout << p_data << " ";
    }

    template <typename T>
    void input(T& p_data){
      std::getline(std::cin, p_data);
    }

    void onClickInput() const;

    void clearScreen() const;
};

#endif // IO_INTERFACE_H
```

System class:

This class is the main controller of the game. It will receive the required data as the input and then creates a match based on that information. These information are the number of players, their name, age, and the sign they choose to play with.

The reason why we separate system and match class was because we wanted to have the option to save multiple games in the future, and separate the process of creating and initializing a match with the process for running it.

The constructor of this class seeds the random number generator with the current time in order to use random algorithms whenever we need to and creates a new match and run it.

Data members:

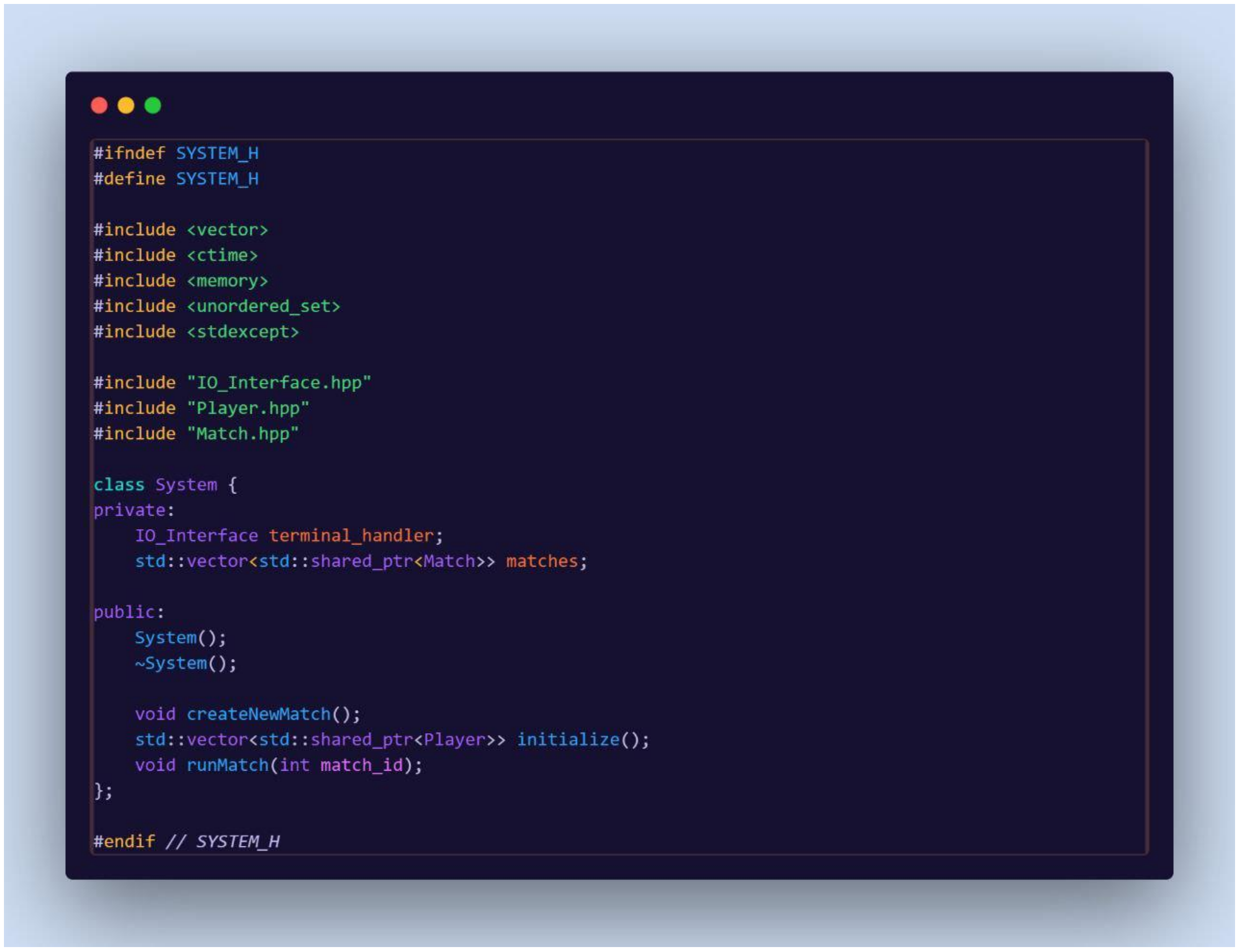
1. terminal_handler: this is the interface used for handling inputs that are coming from command line and print messages and data if whenever we need to.
2. matches: a vector that stores different matches.

Function members:

1. initialize(): this method receives the required data for creating a new match, it reads the data from input(information about

players) and creates a vector that contains shared pointers to Player class and finally returns them. It also validates the inputs from user for different types of exceptions in order to prevent run-time errors.

2. createNewMatch(): creates a new match with the returning data from initialize() and adds it to the matches vector.
3. runMatch(): runs a specific match.



```cpp
#ifndef SYSTEM_H
#define SYSTEM_H

#include <vector>
#include <ctime>
#include <memory>
#include <unordered_set>
#include <stdexcept>

#include "IO_Interface.hpp"
#include "Player.hpp"
#include "Match.hpp"

class System {
private:
    IO_Interface terminal_handler;
    std::vector<std::shared_ptr<Match>> matches;

public:
    System();
    ~System();

    void createNewMatch();
    std::vector<std::shared_ptr<Player>> initialize();
    void runMatch(int match_id);
};

#endif // SYSTEM_H
```

Match class:

Match class represents a match in the game, when we create or select a match in system class a match would begin with the given data about the players.

Data members of this class are the essential data to build and run a match properly. e.g. list of players, lands, adjacent lands, war sign, current season, pass count, terminal handler(instance of IO_Interface) for interacting with players, guide (instance of GameGuide) for using game guidelines, deck of cards, etc.

constructor of this class creates instances of Land and initializes the adjacent list, and finds the youngest player to start the new match by assigning it to the owner of war sign.

Some function members of this class:

1. displayStatus(): this method is used to show game status in three fields:
    a. A list of players with their current score and played cards.
    b. List of lands that each player has captured.
    c. Battle land and season.

2. rechargeDeck(): this method recharges players' hands based on game rules.

3. refreshData(): this method resets data for different battles. E.g. reset current season, pass counter, etc.

4. playerChoice(): this method when player wants to do something in the game. It will get player command as input and handle it for different situations like if it was a card name, the card will be played, or if the player wanted to pass or use help in the game.

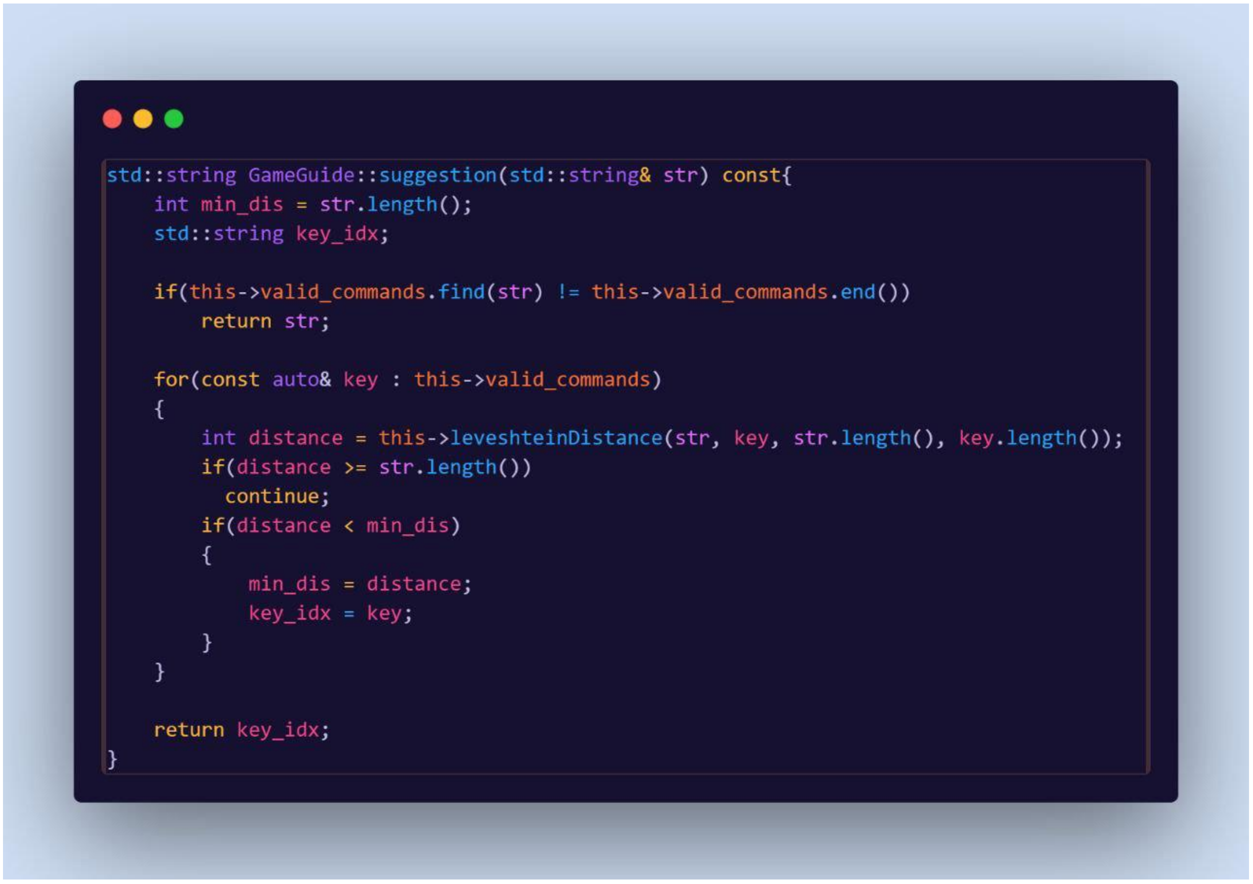
5. run(): this method runs the match in the game until someone wins.

6. war(): war is used for current battle, it handles the players' turns and their actions during the battle, and it will calculate the live score of players in each turn and after someone won the battle, it will call stateWinner method.

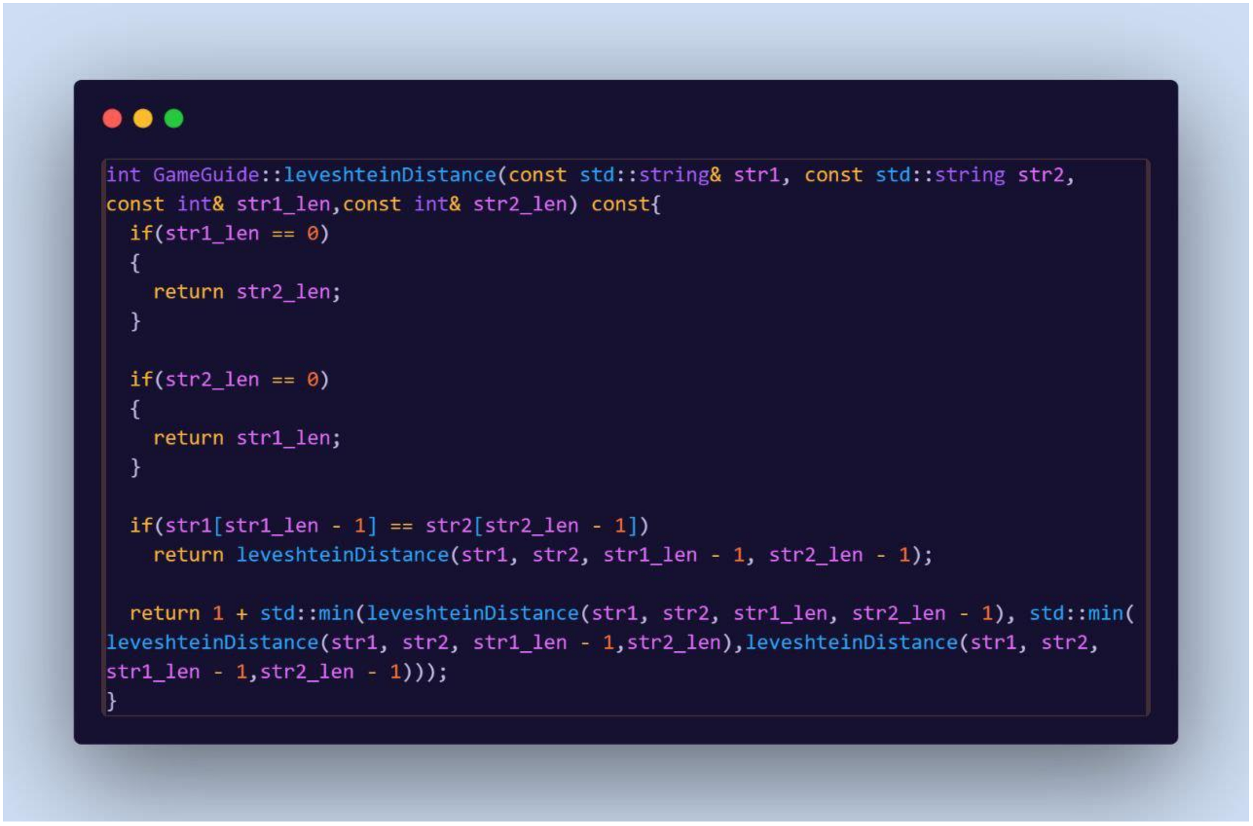
7. stateWinner(): this method is used for assigning the land to winners

8. gameWinner(): this method finds the winner of the game at the end.

9. findStarterPlayer(): This method finds the starter player based on the war sign owner.



```cpp
std::string GameGuide::suggestion(std::string& str) const{
    int min_dis = str.length();
    std::string key_idx;

    if(this->valid_commands.find(str) != this->valid_commands.end())
        return str;

    for(const auto& key : this->valid_commands)
    {
        int distance = this->leveshteinDistance(str, key, str.length(), key.length());
        if(distance >= str.length())
          continue;
        if(distance < min_dis)
        {
            min_dis = distance;
            key_idx = key;
        }
    }

    return key_idx;
}
```

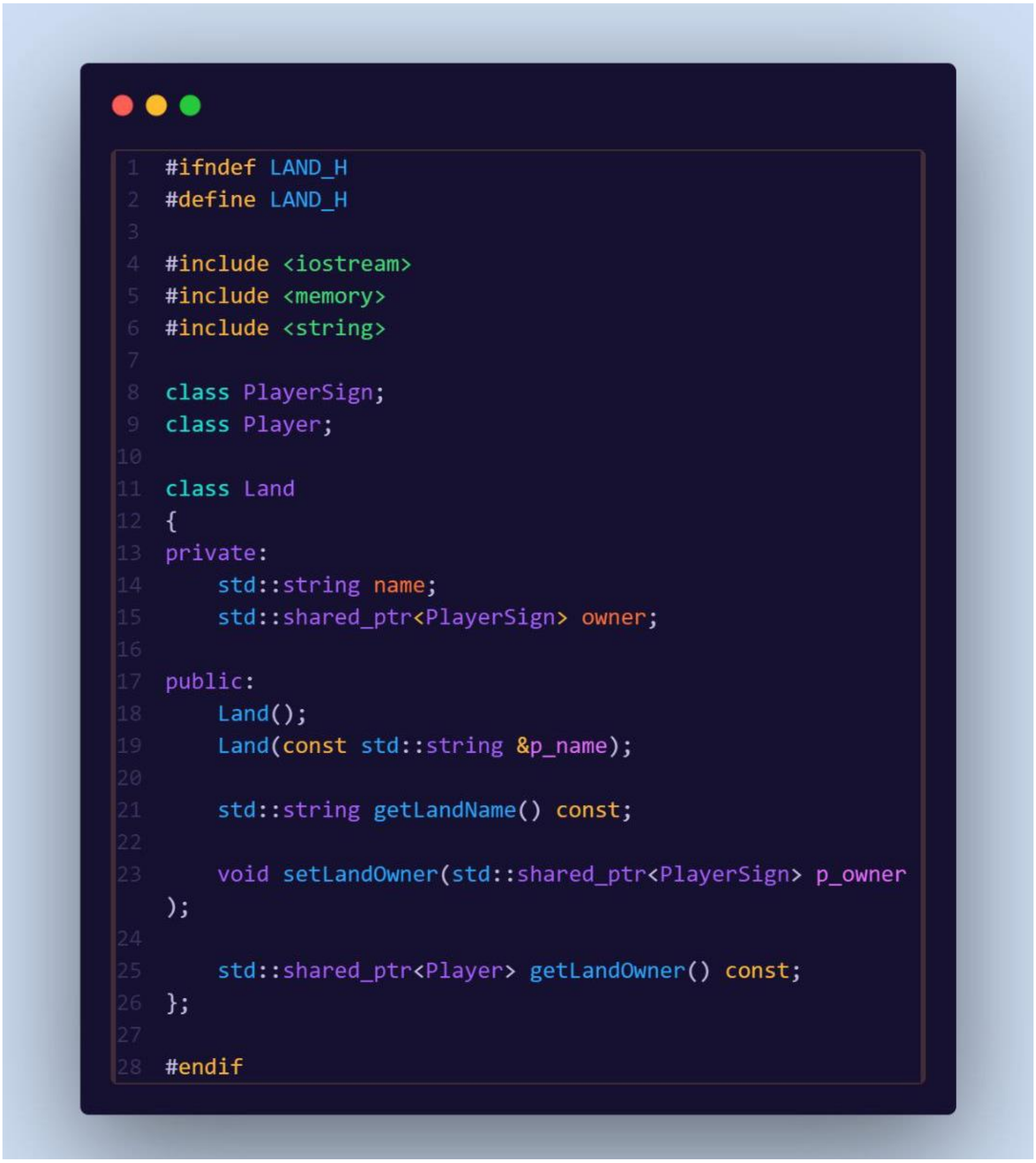


```cpp
int GameGuide::leveshteinDistance(const std::string& str1, const std::string str2,
const int& str1_len,const int& str2_len) const{
  if(str1_len == 0)
  {
    return str2_len;
  }

  if(str2_len == 0)
  {
    return str1_len;
  }

  if(str1[str1_len - 1] == str2[str2_len - 1])
    return leveshteinDistance(str1, str2, str1_len - 1, str2_len - 1);

  return 1 + std::min(leveshteinDistance(str1, str2, str1_len, str2_len - 1), std::min(
leveshteinDistance(str1, str2, str1_len - 1,str2_len),leveshteinDistance(str1, str2,
str1_len - 1,str2_len - 1)));
}
```

Land Class:

This class represents a land in the game. it has name and owner as its data members, and it will set the owner to nullptr in its constructor.

This class only has setters and getters as its function members.



```cpp
#ifndef LAND_H
#define LAND_H

#include <iostream>
#include <memory>
#include <string>

class PlayerSign;
class Player;

class Land
{
private:
    std::string name;
    std::shared_ptr<PlayerSign> owner;

public:
    Land();
    Land(const std::string &p_name);

    std::string getLandName() const;

    void setLandOwner(std::shared_ptr<PlayerSign> p_owner
);

    std::shared_ptr<Player> getLandOwner() const;
};

#endif
```

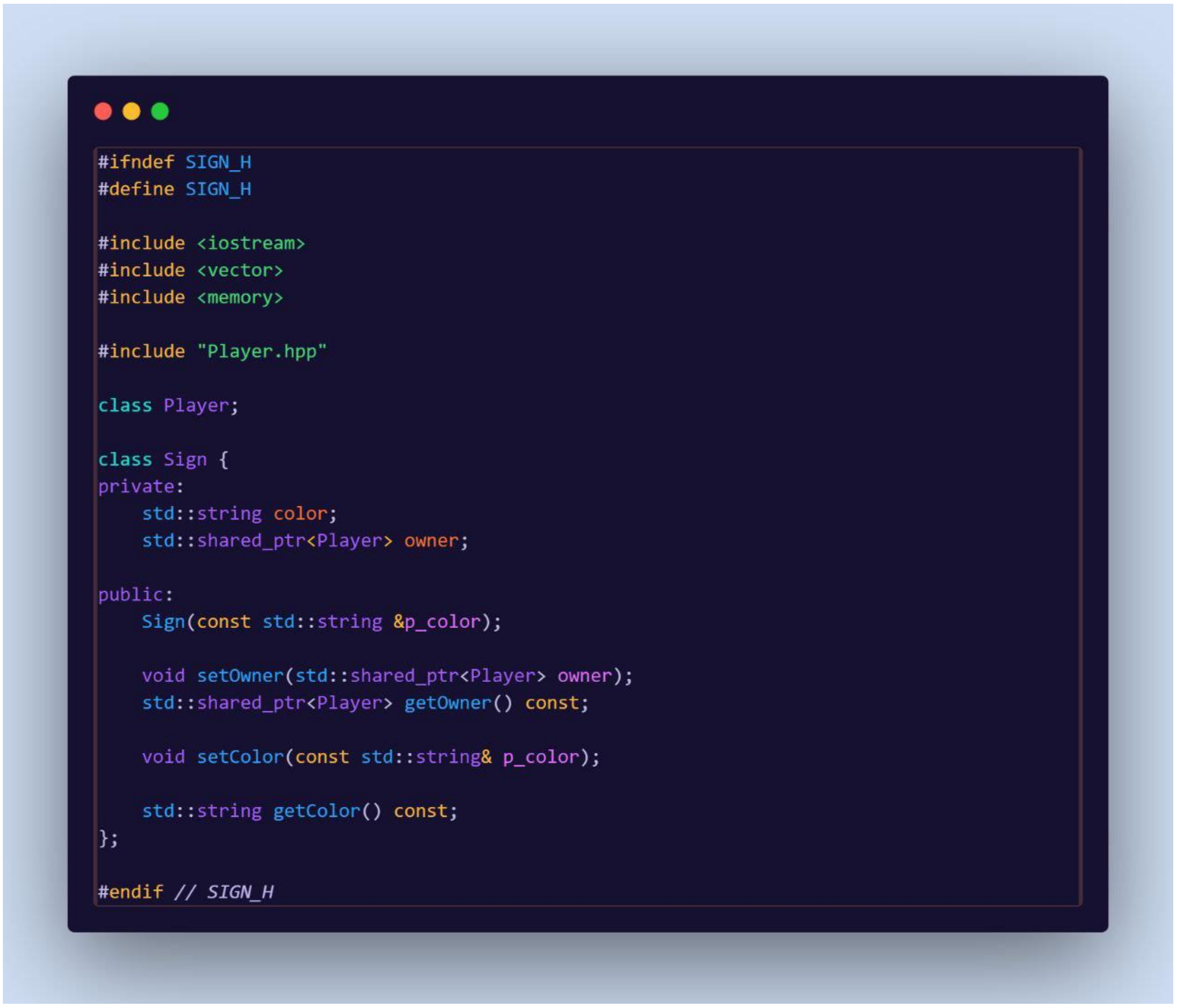
Sign Class:

This class represents signs in the game and it's the parent of PlayerSign and WarSign classes.

This class has color and owner as its data members. In its constructor it will set color to the given color (based on available colors).

Function members of this class are setters and getters of its data members.

This class has aggregation relationship with player class.



```cpp
#ifndef SIGN_H
#define SIGN_H

#include <iostream>
#include <vector>
#include <memory>

#include "Player.hpp"

class Player;

class Sign {
private:
    std::string color;
    std::shared_ptr<Player> owner;

public:
    Sign(const std::string &p_color);

    void setOwner(std::shared_ptr<Player> owner);
    std::shared_ptr<Player> getOwner() const;

    void setColor(const std::string& p_color);

    std::string getColor() const;
};

#endif // SIGN_H
```

PlayerSign Class:

This class is a child of Sign class and represents the signs(marks) that players play with during the game.

It has a vector of Land as its data member and it has a setter and getter for this vector.

This class has aggregation relation with Land class.

WarSign Class:

This class is the other child of Sign class and represents the war sign in the game it has aggregation relationship with Land class. It has a shared pointer to Land class called land.

Function members of this class are a getter and setter for land data member.

Card Class:

This class represents a card in game and it’s the parent class of Special and Soldier classes.

This class has a protected data member which is the name of the card.

This class is an abstract class and it has two pure virtual methods

getCardType() and use().

Special Class:

This is one of the Card children and it is also an abstract class (it doesn’t have an implementation for use method). This class represents Special Cards and 5 classes inherit from this class (Winter, Spring, Scarecrow, Heroine, Drummer)

Childs of this class do not have any additional data member, they just have the implementation of use method.

Soldier Class:

This class is another child of Card. It represents soldiers in game

It has score as data member and it has setter and getter for it.

The constructor of this class inherits from Card constructor and sets the name and score for its instances.

CardDeck Class:

This class is used as the deck of cards in the game. We can shuffle and deal cards to players with this class

There are 2 data members in this class:

1. deck which is the vector of all cards in game .

2. an unordered map that we build deck based on the information in it, it keys card name to the pair of number of that card and its score.

Function members:

1. shuffleCards(): it will shuffle the card deck.
2. generateDeck(): it will generate the card deck based on the unordered map and fills the vector of cards.
3. dealCard(): this method deals cards to the players.
4. Getters and setters

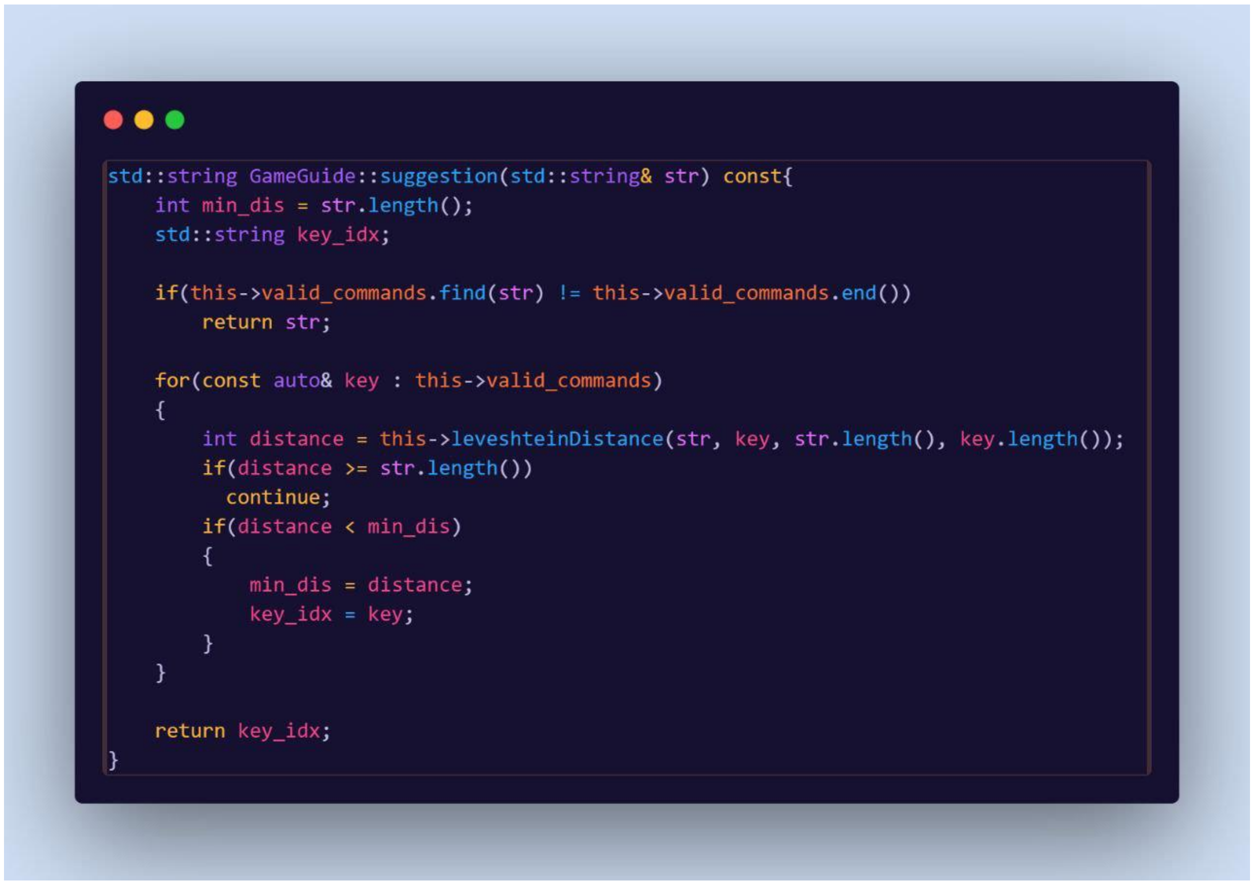
GameGuide Class:

This class is used for working with help commands, in its constructor 3 methods are called to read the data from binary files and fill its data member with them

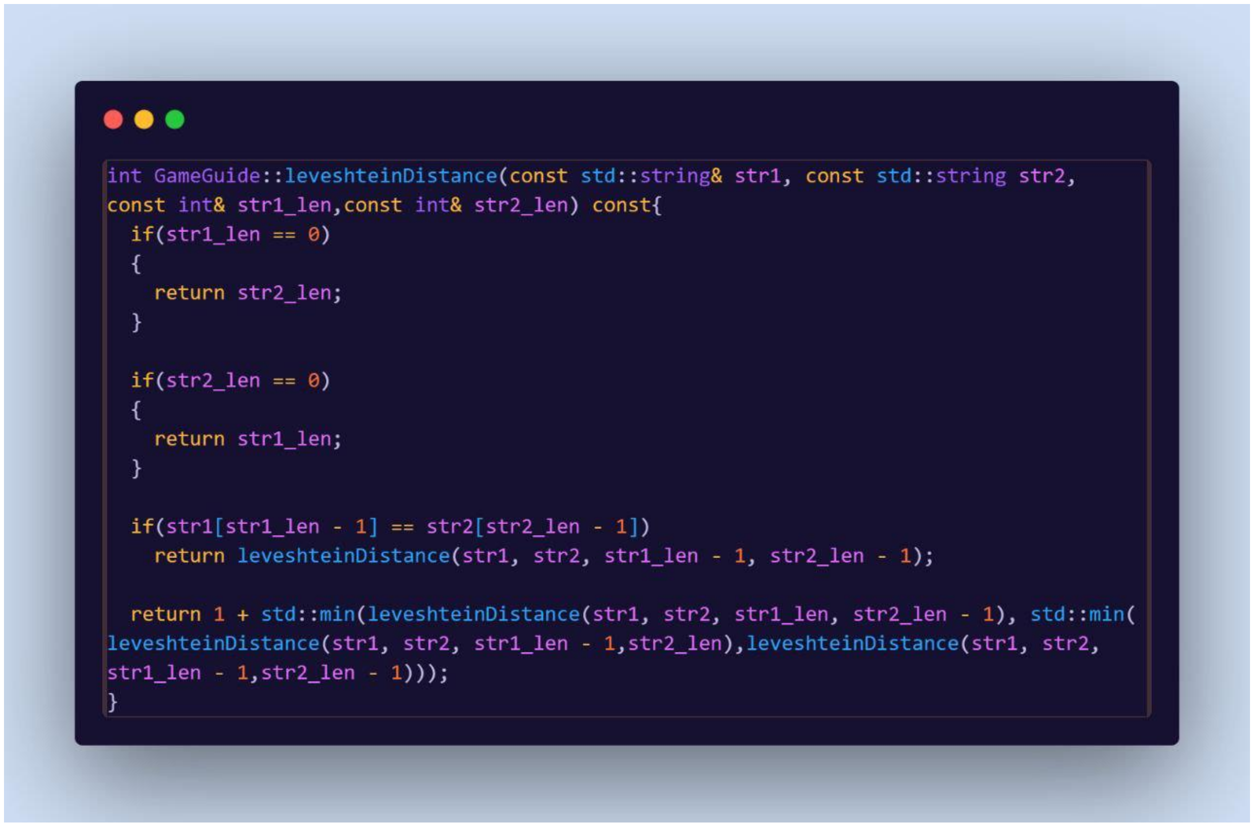
Its data members are terminal_handler (IO_Interface), game rules, valid commands, cards description.

Function members:

1. getters()
2. reading data from binary files methods (readCardInfo, readGameGuide and readValidCommands)
3. leveshteinDistance(): this method uses the levenstein distance algorithm to find the difference between 2 strings
4. suggestion(): this method uses leveshteinDistance() method to autocorrect and suggest the correct words to the player



```cpp
std::string GameGuide::suggestion(std::string& str) const{
    int min_dis = str.length();
    std::string key_idx;

    if(this->valid_commands.find(str) != this->valid_commands.end())
        return str;

    for(const auto& key : this->valid_commands)
    {
        int distance = this->leveshteinDistance(str, key, str.length(), key.length());
        if(distance >= str.length())
          continue;
        if(distance < min_dis)
        {
            min_dis = distance;
            key_idx = key;
        }
    }

    return key_idx;
}
```



```cpp
int GameGuide::leveshteinDistance(const std::string& str1, const std::string str2,
const int& str1_len,const int& str2_len) const{
  if(str1_len == 0)
  {
    return str2_len;
  }

  if(str2_len == 0)
  {
    return str1_len;
  }

  if(str1[str1_len - 1] == str2[str2_len - 1])
    return leveshteinDistance(str1, str2, str1_len - 1, str2_len - 1);

  return 1 + std::min(leveshteinDistance(str1, str2, str1_len, str2_len - 1), std::min(
leveshteinDistance(str1, str2, str1_len - 1,str2_len),leveshteinDistance(str1, str2,
str1_len - 1,str2_len - 1)));
}
```

References:

C++:

1. Effective Modern C++ book by Scott Meyers. (for smart pointers)
2. object oriented analysis and design with applications
3. cpp reference

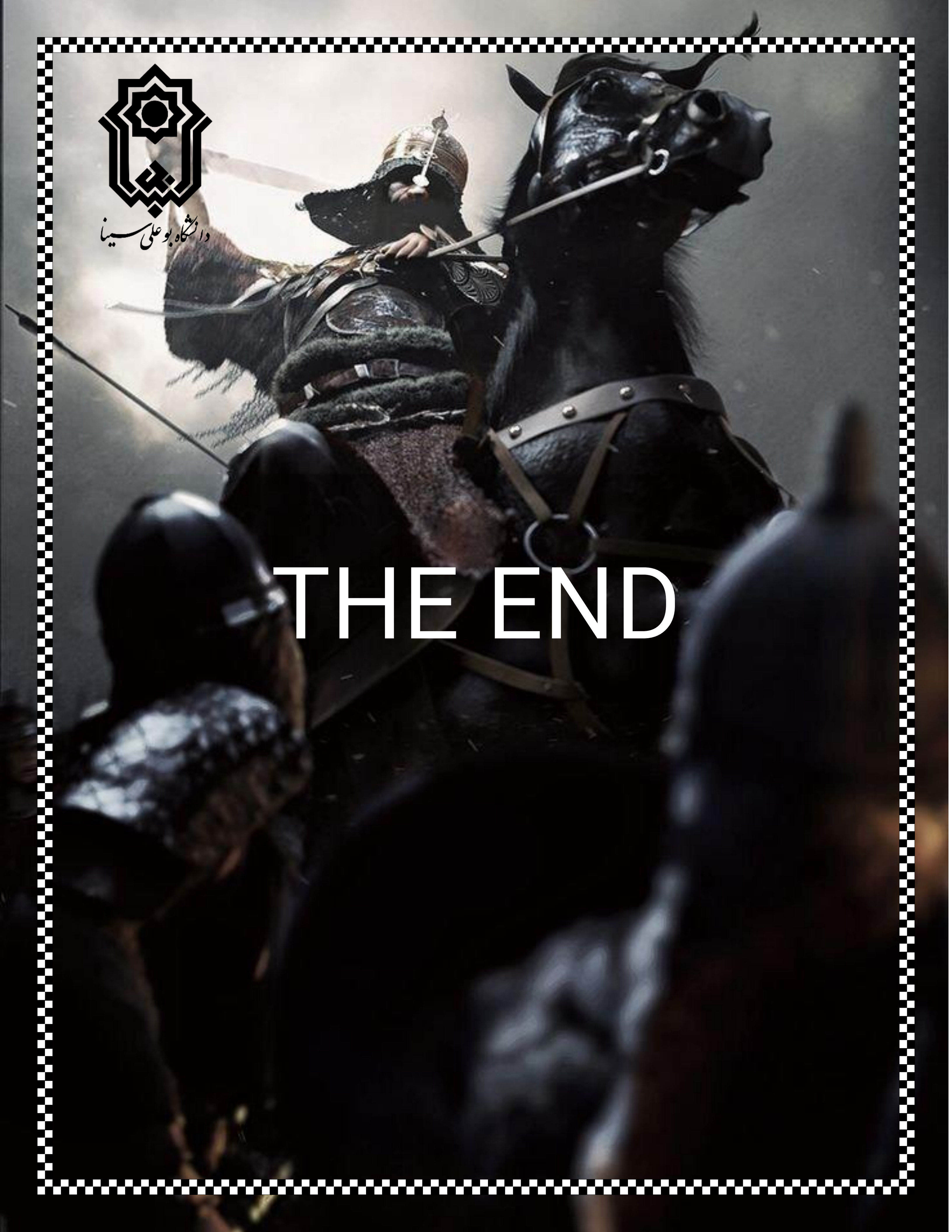
for levenstein distance algorithm:

https://www.baeldung.com/cs/levenshtein-distance-computation

http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata

Linkedin game development rooms

CMake documentation.


دانشگاه بوعلی سینا
THE END