Faculty of engineering and natural sciences Introduction to programming

2022/23.

**Project**

**Game Center**

Student: Amer Jusić 220302210

Faruk Joldić 220302155

Faris Keserović 220302101

Professor: Tarik Namas Ismail Dedić 220302174

Table of content

[1. Main Menu 3](#_Toc304)

[1.1 Overview 3](#_Toc23348)

[1.2 Classes and member functions 4](#_Toc16238)

[2. BlackJack 5](#_Toc11482)

[2.1 Overview 5](#_Toc889)

[2.2 Classes and member functions 6](#_Toc7011)

[3. Horse Race 7](#_Toc22274)

[3.1 Overview 7](#_Toc22011)

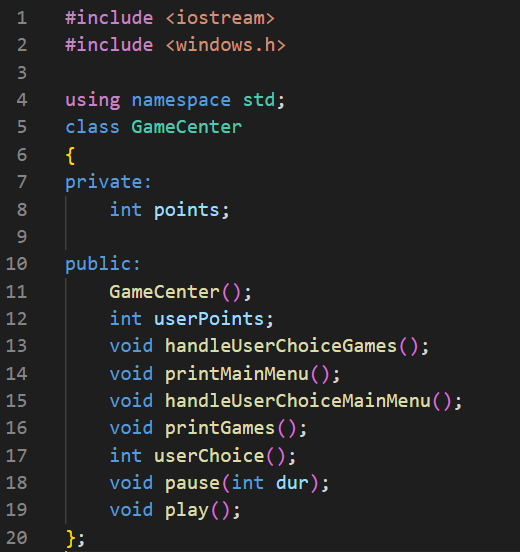
[3.2 Classes and member functions: 8](#_Toc20418)

[4. Minesweeper 9](#_Toc6604)

[4.1 Overview 9](#_Toc830)

[4.2 Classes and Member Functions: 9](#_Toc26405)

# Main Menu



## 1.1 Overview

This code is for a game center that contains several games that users can play. The game center has a main menu that allows users to either play a game or quit the program. If the user chooses to play a game, they are presented with a list of games to choose from, along with the number of points required to play each game. The user can then choose a game to play, and if they have enough points, the game will start and the required number of points will be deducted from their total. If the user wins the game, points are added to their total.

## 1.2 Classes and member functions

The GameCenter class has several member functions that handle various aspects of the game center's functionality:

**GameCenter::GameCenter():** This is the constructor for the GameCenter class. It sets the color of the console text to green, initializes the points member variable to 100, and sets the userPoints member variable to the value of points.

**GameCenter::handleUserChoiceGames():** This function handles the user's choice of which game to play. If the user chooses a game, the function checks if the user has enough points to play the game. If the user has enough points, it starts the game and deducts the required number of points from the user's total points. If the user wins the game, it adds points to the user's total.

**GameCenter::printMainMenu():** This function displays the main menu for the game center, which allows the user to either play a game or quit the program. It then calls the GameCenter::handleUserChoiceMainMenu() function to handle the user's choice.

**GameCenter::handleUserChoiceMainMenu():** This function handles the user's choice from the main menu. If the user chooses to play a game, it calls the GameCenter::printGames() function to display the list of games. If the user chooses to quit, it exits the program.

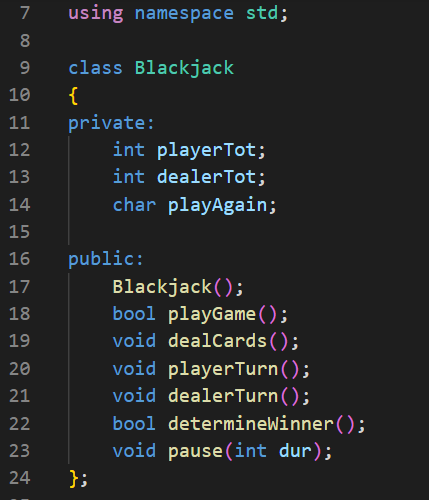
**GameCenter::printGames():** This function displays a list of games that the user can play, along with the number of points required to play each game. It then calls the GameCenter::handleUserChoiceGames() function to handle the user's choice.

**GameCenter::userChoice():** This function prompts the user to enter their choice and returns the choice as an integer.

**GameCenter::pause():** This function is for creating pauses in between main menu printing. This adds to the ease for navigation and a better aesthetics.

**GameCenter::play():** This function only calls the function printMainMenu() with which the game starts.

# BlackJack



## Overview

This code is for a game of Blackjack, a popular card game in which players try to get a total as close to 21 as possible without going over. The goal is to have a higher total than the dealer, or to have the dealer go over 21 (bust). The game is played by dealing two cards to the player and two cards to the dealer, and then allowing the player to take additional cards (hit) or end their turn (stand). The dealer follows a set of rules to decide when to hit or stand. After both the player and dealer have completed their turns, the game compares the totals of the player's and dealer's cards to determine the winner. If either the player or dealer has a total of 21 (referred to as a "blackjack"), they win immediately.

## 2.2 Classes and member functions

The Blackjack class has several member functions that handle various aspects of the game's gameplay:

**Blackjack::Blackjack():** This is the constructor for the Blackjack class. It initializes the playerTot and dealerTot member variables to 0.

**Blackjack::playGame():** This is the main function for playing a round of Blackjack. It deals the initial cards to the player and dealer, checks for blackjacks, and then allows the player to take additional cards or stand. It then allows the dealer to take additional cards according to a set of rules. After both the player and dealer have completed their turns, it calls the Blackjack::determineWinner() function to determine the winner of the game.

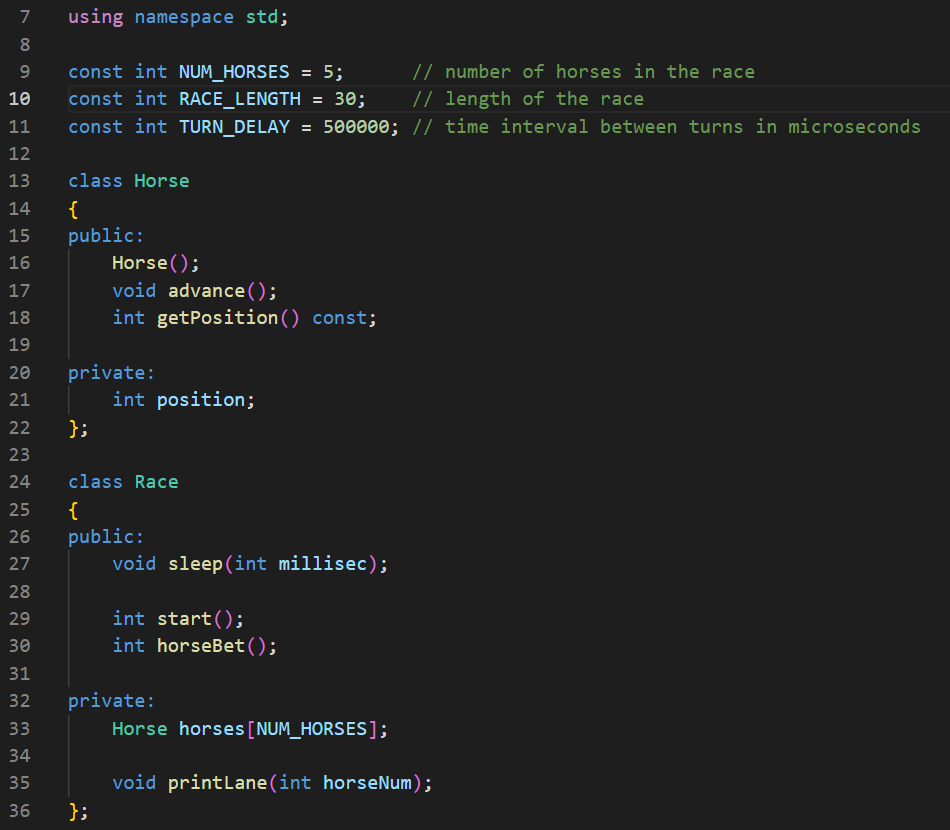
**Blackjack::dealCards():** This function deals two cards to the player and two card.

**Blackjack::playerTurn():** This function allows the player to take additional cards (hit) or end their turn (stand).

**Blackjack::dealerTurn():** This function allows the dealer to take additional cards according to a set of rules.

**Blackjack::determineWinner():** This function compares the totals of the player's and dealer's cards to determine the winner of the game.

# Horse Race



## Overview

This code provides an implementation of a horse racing game. It consists of a Horse class, which has a member variable position that represents the position of the horse on the race track, and member functions advance() and getPosition() which move the horse forward by a random number of steps (0 or 1) and return the horse's current position, respectively. The code also includes a Race class, which has a member function start() that runs a simulated horse race. This function allows the user to place a bet on a particular horse, and then starts the race by advancing the horses along the track until one of them reaches the end. The function then prints out the winner of the race and returns whether the user's bet was correct.

## 3.2 Classes and member functions:

**Horse::Horse():** This is the constructor for the Horse class. It initializes the position member variable to 0.

**Horse::advance():** This member function moves the horse forward by a random number of steps (0 or 1).

**Horse::getPosition():** This member function returns the horse's current position on the track.

**Race::sleep(int millisec):** This member function causes the program to pause for a specified number of milliseconds.

**Race::start():** This is the main function for running a simulated horse race. It allows the user to place a bet on a particular horse, and then starts the race by advancing the horses along the track until one of them reaches the end. It then prints out the winner of the race and returns whether the user's bet was correct.

**Race::printLane(int horseNum):** This member function prints out the lane of a particular horse on the track, with periods representing empty track and the horse's number representing the horse's position.

**Race::horseBet():** This member function allows the user to place a bet on a particular horse. It prompts the user to enter the number of the horse they want to bet on and returns the user's choice.

# Minesweeper



## Overview

The given code is for a game of Minesweeper, a popular puzzle game where players have to locate hidden mines on a board without detonating any of them. The game consists of a grid of squares, each of which may or may not contain a mine. The player is initially presented with the board with some of the squares revealed, and must use this information to deduce the locations of the mines and mark them. The player wins if they are able to successfully mark all the mines on the board before revealing a mine.

## 4.2 Classes and Member Functions:

The Minesweeper class has the following member functions:

**Minesweeper::Minesweeper():** This is a default constructor for the Minesweeper class

**Minesweeper::printBoard():** This function takes a 2D char array as input and prints out the current state of the board to the console.

**Minesweeper::userMove():** This function prompts the user to enter the row and column numbers of the square they want to reveal and stores the input in two integer variables.

**Minesweeper::validPos():** This function takes an integer row and column number as input and returns a boolean value indicating whether the given position is within the boundaries of the board.

**Minesweeper::placeMine():** This function takes a 2D int array and a 2D char array as input and randomly places mines in the char array.

**Minesweeper::isMine():** This function takes an integer row and column number and a 2D char array as input and returns a boolean value indicating whether there is a mine at the given position.

**Minesweeper::countMines():** This function takes an integer row and column number, a 2D int array, and a 2D char array as input and returns the number of mines in the 8 adjacent squares to the given position.

**Minesweeper::userTurn():** This function handles the player's turn in the game, including prompting the user for their move and updating the board accordingly.

**Minesweeper::initial():** This function initialises the board that the user sees and the original one.

**Minesweeper::playUntil():** This is the most complex function of all. The function first checks if the user has already revealed the cell at the given row and column. If the cell has already been revealed, the function returns false. If the cell has not been revealed, the function checks if the cell at the given row and column contains a mine. If it does, the function reveals all mines on the board to the user, prints the board, and outputs a message indicating that the user has lost the game. If the cell does not contain a mine, the function calculates the number of mines present in the cells adjacent to the chosen cell and decrements the number of moves the user has left by 1. The function then reveals the chosen cell to the user by replacing its value with the number of adjacent mines.

If the number of adjacent mines is 0, the function then recursively applies itself to each of the 8 cells surrounding the chosen cell, as long as they are valid positions on the board (i.e. within the bounds of the board) and do not contain mines.