SOFE 3950U: Operating Systems

Tutorial #4: Jeopardy

Tutorial CRN: 74027
Tutorial Group 2

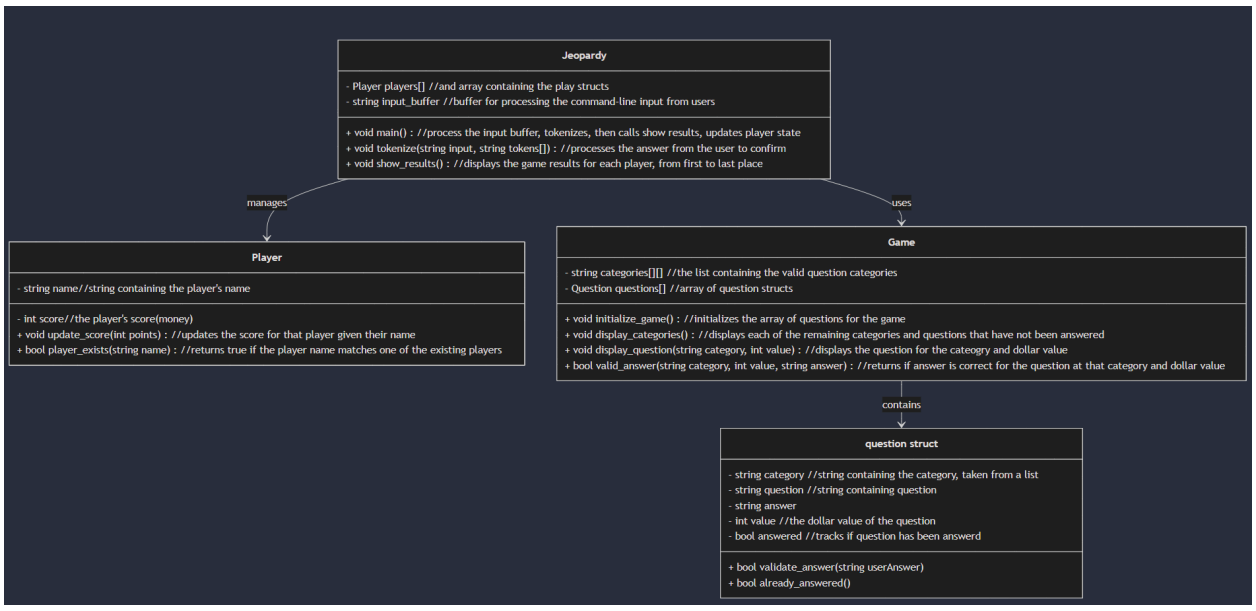
Rivka Sagi	100780926
Rhea Mathias	100825543
Julian Olano Medina	100855732
Omokorede Olobayo	100875182

Introduction

In this tutorial, we built a simple command-line version of Jeopardy. It is a simple game where a player picks a category and dollar value and then answers some questions. Our version of Jeopardy allows four players to take turns picking questions from different categories, answering them, and earning or losing points based on whether they are correct or not. The game keeps track of scores and determines a winner at the end.

Game Functionalities

1. Players enter their names (this is used for tracking the scores).
2. The game displays a table of categories and dollar values.
3. A player selects a question, and the program checks to see if that question has already been answered.
4. The question they selected is displayed, and the player must respond in the format of "What is" or "Who is."
5. If the answer is correct, the player earns some points. If not, the correct answer is put on display, and they get no points.
6. The game continues until all questions are answered.
7. At the end of the game, all the scores are displayed, and the player with the highest score wins.



Code

In order to fulfill all the game functionalities and requirements, the game is divided into different files:

- Player.c and player.h: deals with the player initialization and keeps track of both the player roster and the points associated with each player
- Question.c and question.h: deals with all the functionalities related with the questions such as creating the categories, points and the questions themselves. There are also functions that create the question grid with their values displayed along with showing which questions were already answered
- Jeopardy.c and jeopardy.h: uses the main() function to render inputs from users as well as tokenize the inputs and display the results of each round.

Players.c

All variables are initialized, and the methods are called in the players.h file.

```
// Returns true if the player name matches one of the existing players
bool player_exists(player *players, int num_players, char *name)
{
    for (int i=0; i<num_players; i++) {
        if (strncasecmp(players[i].name, name) == 0) {
            return true;
        }
    }
    return false;
}
```

The **player_exists()** function checks if a player's name matches a list of existing players by looping through the names and by using **strncasecmp** to compare the player's name (case insensitive) to the list, it returns true if the name exists or returns false if it does not.

```c
// Go through the list of players and update the score for the
// player given their name
void update_score(player *players, int num_players, char *name, int score)
{
    for (int i=0; i<num_players; i++){
        if (strncasecmp(player[i].name, name) == 0){
            players[i].score = score;
            return 0;
        }
    }
}
```

The **update_score()** function updates a player's score by looping through the list of players to find a player's name (using **strncasecmp**), and once the name is found, it updates that player's score.

Questions.c

```c
// Initializes the array of questions for the game
void initialize_game(void)
{
    //first allocate memory for the questions array,
    //in our project we are using 4 categories, so there are 16 questions total
    question questions[16];

    //initialize questions, 4 per category:

    //first category: movies
    strcpy(questions[1].category, "animated movies");
    strcpy(questions[1].question,"a movie about an ogre and a donkey who go on an adventure to save a swamp");
    strcpy(questions[1].answer, "Shrek");
    questions[1].value = 100; //lowest level question
    questions[1].answered = false;

```

The **initialize_game()** code creates an array of question structs to hold the questions and then assigns the category, question, answer, value, and answered status to the structs initialized in the array.

```c
// Returns true if the answer is correct for the question for that category and dollar value
bool valid_answer(char *category, int value, char *answer)
{
    bool answer;
    //lookup the question to compare the answer
    for(int i=0; i<16; i++){
        if (category == questions[i].category && value == questions[i].value){
            // use the strncasecmp to check if the answer is correct while disregarding case sensitivity
            answer = (strncasecmp(questions[i].answer, answer) == 0)? true: false;
        }
    }
    return answer;
}
```

The **valid_answer()** code finds the relevant question in the array by searching for the category and the question value. It then compares the provided answer with the correct answer, returning true if the answer is correct and false if not.

```
149  // Returns true if the question has already been answered
150  bool already_answered(char *category, int value)
151  {
152      // lookup the question and see if it's already been marked as answered
153      for(int i=0; i<16; i++){
154          if (category == questions[i].category && value == questions[i].value){
155              //return the status of the question
156              return questions[i].answered;
157          }
158      }
159  }
160
```

The **already_answered()** code finds the relevant question in the array by searching for the category and the question value. It then returns the answered status of the question, whether it has already been answered or not.

Jeopardy.c

```
void tokenize(char *input, char **tokens){

    // tokenizes the input by spaces
    *tokens = strtok(input, " "); //Tokenizes "what" or "who"
    *tokens = strtok(NULL, " "); // tokenizes "is"
    *tokens = strtok(NULL, " "); // tokenizes the actual answer
```

The **tokenize()** function is used to tokenize only the answer from the input by disregarding the “what is” or “who is” statement considering that the answer is only one word. To do so, **strtok()** is used to tokenize each word of the input by using a space as a delimiter.

```c
// Displays the game results for each player, their name and final score, ranked from first to last place
void show_results(player *players, int num_players){
    //checks for the max score by storing max score in a variable and swapping if another max is found using selection sort
    for (int i=0; i <num_players -1; i++){
        int max = i; //stores current player as max
        for (int j= i+1; j < num_players; j++){
            if (players[j].score > players[max].score){
                max = j;//stores the player after i as max if it has a greater score
            }
        }
        // swaps player into "hold"
        if (max != i){
            players hold= player[i];
            players[i] = player[max];
            players[max] = hold;

        }
    }

    //displays player result
    printf("Player scoreboard: \n");
    for (int i=0; i< num_players; i++){
        printf("%s: %d \n", players[i].name, players[i].score);
    }
};
```

The **show_results()** function is used to display the current player scoreboard by iterating through the player roster and comparing their score value by using the selection sort algorithm to display the player name and score in descending order (highest player score on top).

Gitub Link:
https://github.com/RivkaRSagi/OSTutorial4.git

Conclusion

We were able to successfully create a functional Jeopardy game in C. We structured the program with multiple source files so as to allow four players to compete, keep track of their scores, and then determine a winner based on the highest points earned.