FAST NUCES Programming Olympics

PROGRAMMING FUNDAMENTAL CS1002 - Fall 2024

Release Date: 15th November 2024 Due Date: 29th november 2024

Total Marks 70

Congratulations, athletes! You've advanced to the next stage of the Programming Olympics for Fall 2024 by excelling in Assignment 2. Your journey in CS1002 continues with the third round of challenges.

- The rules remain simple: Your mission is to cross the finish line, but not just with speed. This time, we focus even more on elegance, creativity, and the dedication you put into your problem-solving journey.
- True champions face obstacles with integrity. Remember, the biggest challenge isn't just solving the problems—it's overcoming the temptations of using shortcuts like ChatGPT, Gemini, or relying on others' solutions. Such actions will lead to immediate disqualification.

Now, show us your best work, stay honest, and keep your eyes on the prize. Let the challenge begin!

How to submit

There are a total of 7 questions in the assignment and each question carries equal marks i.e 10. You must submit the .c files compressed in a zip folder with your student id on google classroom within the due date.

Question 01

Write a program that contains a structure named Employee which contains the following data members:

- employeeCode
- employeeName
- dateOfJoining

Perform the following operations:

1. **Define a function** that assigns user-defined values to these variables.
2. **Create an array of 4 Employee structs** and initialize them with user-defined values.
3. **Define a function** that asks the user to enter the current date, calculates the tenure of each employee, and checks if the tenure is more than three years. Display the details of employees with tenure of more than three years and the count of such employees.

```
#include <stdio.h>
#include <stdbool.h>

// Define the Date structure
typedef struct {
    int year;
    int month;
    int day;
} Date;

// Define the Employee structure
typedef struct {
    int employeeCode;
    char employeeName[50];
    Date dateOfJoining;
} Employee;

// Helper function to validate the month
bool isValidMonth(int month) {
    return month >= 1 && month <= 12;
}

// Helper function to validate the day based on the month and year
bool isValidDay(int day, int month, int year) {
    int daysInMonth[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

```c
    // Check for leap year in February
    if (month == 2 && ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)) {
        daysInMonth[1] = 29;
    }

    return day >= 1 && day <= daysInMonth[month - 1];
}

// Function to assign values to an array of employees
void assignEmployee(Employee emp[], int size) {
    for (int i = 0; i < size; i++) {
        printf("\nEnter Employee_%d's Code: ", i + 1);
        scanf("%d", &emp[i].employeeCode);

        printf("Enter Employee_%d's Name: ", i + 1);
        scanf("%49s", emp[i].employeeName);

        // Input and validate date of joining
        while (1) {
            printf("Enter Employee_%d's Date of Joining (Year Month Day): ", i + 1);
            scanf("%d %d %d", &emp[i].dateOfJoining.year,
                         &emp[i].dateOfJoining.month,
                         &emp[i].dateOfJoining.day);

            if (isValidMonth(emp[i].dateOfJoining.month) &&
                isValidDay(emp[i].dateOfJoining.day, emp[i].dateOfJoining.month,
emp[i].dateOfJoining.year)) {
                break; // Valid date
            } else {
                printf("Invalid date entered, please re-enter.\n");
            }
        }
    }
}

// Function to calculate tenure and check if more than 3 years
void calculateTenure(Employee employees[], int size) {
    int currentYear, currentMonth, currentDay;
    printf("Enter the current date (YYYY MM DD): ");
    scanf("%d %d %d", &currentYear, &currentMonth, &currentDay);

    int count = 0;
    printf("\nEmployees with tenure more than 3 years:\n");
    printf("Code\tName\t\tDate of Joining\t\tTenure (Years)\n");
    printf("----------------------------------------------------\n");

    for (int i = 0; i < size; i++) {
        int dojYear = employees[i].dateOfJoining.year;
        int dojMonth = employees[i].dateOfJoining.month;
        int dojDay = employees[i].dateOfJoining.day;

        // Calculate tenure
        int tenureYears = currentYear - dojYear;
        if (currentMonth < dojMonth || (currentMonth == dojMonth && currentDay <
dojDay)) {
            tenureYears--;
        }

        // Check and display if tenure is more than 3 years
        if (tenureYears > 3) {
            printf("%d\t%-15s%d-%02d-%02d\t%d\n", employees[i].employeeCode,
                   employees[i].employeeName,
                   dojYear, dojMonth, dojDay, tenureYears);
            count++;
        }
    }

    printf("\nTotal employees with tenure more than 3 years: %d\n", count);
}

int main() {
    const int numEmployees = 4;
    Employee employees[numEmployees];

    // Collect employee data
    assignEmployee(employees, numEmployees);

    // Calculate tenure and display results
    calculateTenure(employees, numEmployees);

    return 0;
}
```

Question 02

Write a program that organizes a digital cricket match, "Cricket Showdown," where two players, Player 1 and Player 2, compete over 12 balls. Each player takes turns to score runs on each ball. Players can enter scores between 0 and 6 for each ball, and if a score outside this range is entered, it will be disregarded, but the ball will still be marked.

1. **Define a structure Player** with the following members:
    a. ballScores[12]: An array to store the score for each ball.
    b. playerName: A string to hold the player's name.
    c. totalScore: An integer to store the total score for each player.
2. **Define functions**:
    a. playGame(struct Player player): This function prompts each player to enter their score for each of the 12 balls.
    b. validateScore(int score): This function checks if the score is between 0 and 6 (inclusive). If it's not, the score is disregarded, but the ball is still marked.
    c. findWinner(struct Player player1,struct Player player2): Determines the winner based on the total score.
    d. displayMatchScoreboard(struct Player player1,struct Player player2): Displays a summary of each player's performance, including each ball's score, the average score, and total score.

```
 #include<stdio.h>
struct Player{
        int ballScores[12];
        char playerName[30];
        int totalScore;
};
int validateScore(int score){
        if(score>=0 && score<=6)
        return 1;
        else return 0;
}
void playGame(struct Player *player){
        int score;
        for(int i=0; i<12; i++){
                printf("Enter score for ball %d : ",i+1);
                scanf(" %d",&score);
                if(validateScore(score)) {
                        player->ballScores[i] = score;
                        player->totalScore += score;
                }else {
```

```
                printf("Invalid score! This ball is marked but score is disregarded.\n");
                player->ballScores[i]=0;
            }
        }
}
void findWinner(struct Player player1,struct Player player2){
        if(player1.totalScore > player2.totalScore) printf("Congratulations! to %s, Won the
game",player1.playerName);
        else if(player1.totalScore < player2.totalScore) printf("Congratulations! to %s, Won the
game",player2.playerName);
        else if(player1.totalScore == player2.totalScore) printf("Match drawn!");
}
void displayMatchScoreboard(struct Player player1,struct Player player2){
        printf("\n\nMatch Score Board: \n\n");
        printf("%-15s","Player");
        for(int i=0;i<12;i++) printf("B%-5d",i+1);
        printf("\tTotal\tAverage\n");
        printf("-------------------------------------------------------------------------------------\n");
        printf("%-15s",player1.playerName);
        for(int i=0;i<12;i++) printf("%-6d",player1.ballScores[i]);
        printf(" %d\t%.2f",player1.totalScore,player1.totalScore/12.0);
        printf("\n");
        printf("%-15s",player2.playerName);
        for(int i=0;i<12;i++) printf("%-6d",player2.ballScores[i]);
        printf(" %d\t%.2f",player2.totalScore,player2.totalScore/12.0);
        printf("\n-------------------------------------------------------------------------------------\n");

}
int main(){
        struct Player player1,player2;
        printf("Enter Name of player 1: ");
        gets(player1.playerName);
        printf("Enter Name of player 2: ");
        gets(player2.playerName);
        player1.totalScore=0;
        player2.totalScore=0;
        printf("\nPlayer %s's turn: \n",player1.playerName);
        playGame(&player1);
        printf("\nPlayer %s's turn: \n",player2.playerName);
        playGame(&player2);

        displayMatchScoreboard(player1,player2);
        printf("\n\nWinner announcement:\n");
        findWinner(player1,player2);
        return 0;}
```

Question 03

Create a program to validate an email address based on a few basic criteria. The program will prompt the user to enter an email address and will dynamically allocate memory to store and process the input.

Define the following function:

1. *int validateEmail(char* email)*: This function validates the email based on the following criteria:
    - Contains exactly one @ symbol.
    - Contains at least one dot (.) after the @ symbol.
    - Is non-empty.
    - Returns 1 if the email is valid, and 0 if invalid.

Steps:

1. User Input: Prompt the user to enter an email address. Allocate memory dynamically for the email, ensuring the memory size is based on the input length.
2. Validation Process:
    - Call the validateEmail function to check if the email meets the criteria.
3. Display Results:
    - Print "Valid Email" if the email meets the criteria.
    - Print "Invalid Email" if the email does not meet the criteria.
4. Memory Cleanup:
    - Free the dynamically allocated memory after validation to prevent memory leaks.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

 int validateEmail(char* email){
   if(email == NULL || strlen(email) == 0)
      return 0;
         int counterAt=0, counterDot=0;
   for (int i = 0; i < strlen(email); i++) {
      if (email[i] == '@') {
         counterAt++;
      }
      else if (counterAt == 1 && email[i] == '.') {
         counterDot = 1;
```

```
        }
      }
      return (counterAt == 1 && counterDot==1);
}

int main(){
    int len=100; //initial buffer
            char *email = (char*)malloc(len*sizeof(char));
    if(email==NULL){
        printf("Memory allocation failed!!\n");
        return 1;//main will return or exit
    }
    printf("Enter the email address: ");
    getchar();
    fgets(email, len, stdin);

    // email[strlen(email)-1]='\0'; //remove newline
    // Remove the newline character if it exists
    int emailLen = strlen(email);
    if (emailLen > 0 && email[emailLen-1] == '\n') {
        email[emailLen - 1] = '\0';
    }
    // DMA - memory size is based on the input length
    char *resizedEmail = realloc(email, (emailLen) * sizeof(char));
    if (resizedEmail == NULL) {
        printf("Memory reallocation failed!!\n");
        free(email); // Free the original memory to avoid leaks
        return 1;
    }
    email = resizedEmail;

    //validate email
    if(validateEmail(email))
        printf("Valid Email\n");
    else
        printf("Invalid Email\n");

    //deallocate memory
    free(email);
            return 0;
}
```

Question 04

You are creating a system to track employee performance ratings over multiple evaluation periods. The system will dynamically allocate memory to store ratings and perform various analysis tasks, including finding the top-performing employee and the best-rated evaluation period.

Define the Employee Structure:

Create a structure called Employee with the following fields:

- ratings: A dynamically allocated array to store the ratings for each evaluation period.
- totalScore: An integer to store the employee's total score across all evaluation periods.

Define the following function:

1. Input Ratings Function:
    - Implement a function ***void inputEmployees(int** ratings, int numEmployees, int numPeriods)*** to allow the user to enter ratings for each employee across all evaluation periods.
    - Each rating should be between 1 and 10 (inclusive). Input validation should be implemented to enforce this.
2. Display Performance Function:
    - Implement a function ***void displayPerformance(int** ratings, int numEmployees, int numPeriods)*** that displays the performance ratings for each employee across all evaluation periods.
3. Find Employee of the Year Function:
    - Implement a function ***int findEmployeeOfTheYear(int** ratings, int numEmployees, int numPeriods)*** to calculate and return the index of the employee with the highest average rating.
4. Find Highest Rated Period Function:
    - Implement a function ***int findHighestRatedPeriod(int** ratings, int numEmployees, int numPeriods)*** to calculate and return the evaluation period with the highest average rating across all employees.
5. Find Worst Performing Employee Function:
    - Implement a function ***int findWorstPerformingEmployee(int** ratings, int numEmployees, int numPeriods)*** to calculate and return the index of the employee with the lowest average rating.

Memory Management:

- Dynamically allocate memory for each employee's ratings based on the number of evaluation periods.
- After completing all tasks, ensure that the dynamically allocated memory is properly freed to prevent memory leaks.

```
#include <stdio.h>
```

```
#include <stdlib.h>

struct Employee{
    int** ratings;
    int numEmployees;
    int numPeriods;
} ;


// Function to input ratings
void inputEmployees(int*** ratings, int numEmployees, int numPeriods) {
    for (int i = 0; i < numEmployees; i++) {
        printf("Enter ratings for Employee %d (between 1 and 10):\n", i + 1);
        for (int j = 0; j < numPeriods; j++) {
            do {
                printf("  Period %d: ", j + 1);
                scanf("%d", &(*ratings)[i][j]);
                if ((*ratings)[i][j] < 1 || (*ratings)[i][j] > 10) {
                    printf("Invalid input! Ratings must be between 1 and 10.\n");
                }
            } while ((*ratings)[i][j] < 1 || (*ratings)[i][j] > 10);
        }
    }
}

// Function to display performance
void displayPerformance(int** ratings, int numEmployees, int numPeriods) {
    printf("\nPerformance Ratings:\n");
    for (int i = 0; i < numEmployees; i++) {
        printf("Employee %d: ", i + 1);
        for (int j = 0; j < numPeriods; j++) {
            printf("%d\t", ratings[i][j]);
        }
        printf("\n");
    }
}

// Function to find Employee of the Year
int findEmployeeOfTheYear(int** ratings, int numEmployees, int numPeriods) {
    int maxIndex = 0;
    double maxAverage = 0;
    for (int i = 0; i < numEmployees; i++) {
        int total = 0;
        for (int j = 0; j < numPeriods; j++) {
            total += ratings[i][j];
        }
        double average = (double)total / numPeriods;
        if (average > maxAverage) {
            maxAverage = average;
            maxIndex = i;
        }
    }
    return maxIndex;
}

// Function to find Highest Rated Period
int findHighestRatedPeriod(int** ratings, int numEmployees, int numPeriods) {
    int maxIndex = 0;
    double maxAverage = 0;
    for (int j = 0; j < numPeriods; j++) {
        int total = 0;
        for (int i = 0; i < numEmployees; i++) {
            total += ratings[i][j];
        }
        double average = (double)total / numEmployees;
        if (average > maxAverage) {
            maxAverage = average;
            maxIndex = j;
        }
    }
    return maxIndex;
}

// Function to find Worst Performing Employee
int findWorstPerformingEmployee(int** ratings, int numEmployees, int numPeriods) {
    int minIndex = 0;
    double minAverage = 1e9; // Set to a large value
    for (int i = 0; i < numEmployees; i++) {
        int total = 0;
        for (int j = 0; j < numPeriods; j++) {
            total += ratings[i][j];
        }
        double average = (double)total / numPeriods;
        if (average < minAverage) {
            minAverage = average;
            minIndex = i;
        }
    }
    return minIndex;
}


int main() {
    struct Employee emp;

    printf("Enter the number of employees: ");
    scanf("%d", &emp.numEmployees);
    printf("Enter the number of evaluation periods: ");
    scanf("%d", &emp.numPeriods);

    // Allocate memory for ratings (2D array)
    emp.ratings = (int**)malloc(emp.numEmployees * sizeof(int*));
    if (emp.ratings == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
    for (int i = 0; i < emp.numEmployees; i++) {
        emp.ratings[i] = (int*)malloc(emp.numPeriods * sizeof(int));
        if (emp.ratings[i] == NULL) {
            printf("Memory allocation failed!\n");
            return 1;
        }
    }

    // Input ratings
    inputEmployees(&emp.ratings, emp.numEmployees, emp.numPeriods);

    // Display ratings
    displayPerformance(emp.ratings, emp.numEmployees, emp.numPeriods);

    // Find and display Employee of the Year
    int employeeOfYear = findEmployeeOfTheYear(emp.ratings, emp.numEmployees,
emp.numPeriods);
    printf("Employee of the Year: Employee %d\n", employeeOfYear + 1);

    // Find and display Highest Rated Period
    int highestRatedPeriod = findHighestRatedPeriod(emp.ratings, emp.numEmployees,
emp.numPeriods);
    printf("Highest Rated Period: Period %d\n", highestRatedPeriod + 1);

    // Find and display Worst Performing Employee
    int worstEmployee = findWorstPerformingEmployee(emp.ratings, emp.numEmployees,
emp.numPeriods);
    printf("Worst Performing Employee: Employee %d\n", worstEmployee + 1);

    // Free allocated memory
    for (int i = 0; i < emp.numEmployees; i++) {
        free(emp.ratings[i]);
    }
    free(emp.ratings);

    return 0;
}
```

Question 05

You're building an inventory system for a pet shop called "**Pets in Heart**" that keeps track of different species of animals and their specific supplies (e.g., food, toys, bedding). The shop inventory system uses a 2D dynamic array char** speciesSupplies where:

- Rows are explicitly set for each species (e.g., "Dogs," "Cats," "Parrots"), and each row corresponds to a different species.
- Columns are dynamically allocated for each species to hold their specific supplies (e.g., "Food," "Leash," "Toys").

Task:

Write a C program that:

1. Initialize the Inventory: Allocate memory for a specified number of species, with each species having its own list of supplies (initially empty).
2. Add Supplies: For each species, dynamically allocate memory for a list of supplies and allow the user to add supplies for that species.
3. Update Supplies: Let users update the name of a supply item for a specific species.
4. Remove Species: Allow users to delete a species and free the associated memory (both for the species and its supplies).
5. Display Inventory: Show the current supplies for each species in the inventory.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define INITIAL_SUPPLY_CAPACITY 5
#define MAX_SPECIES 3

// Define string as a typedef for char *
```

```
typedef char* string;

const char* columnsName[] = {
   "Dogs",
   "Cats",
   "Parrots"
};

int numSpecies = 3; // Number of species
int* numSupplies;   // Array to track the number of supplies for each species
string** speciesSupplies; // 2D dynamic array to store species supplies

// Initialize inventory
void initializeInventory() {
   numSupplies = (int*)malloc(numSpecies * sizeof(int)); // Allocate memory for supply counts
   speciesSupplies = (string**)malloc(numSpecies * sizeof(string*)); // Allocate memory for the
rows (species)

   for (int i = 0; i < numSpecies; i++) {
      numSupplies[i] = 0; // Initially no supplies for any species
      speciesSupplies[i] = (string*)malloc(INITIAL_SUPPLY_CAPACITY * sizeof(string)); //
Allocate memory for supplies for each species
   }
}

// Add a supply to a species
void addSupply(int speciesIndex, const char* supply) {
   speciesSupplies[speciesIndex][numSupplies[speciesIndex]] = (string)malloc(strlen(supply) +
1);
   strcpy(speciesSupplies[speciesIndex][numSupplies[speciesIndex]], supply);
   numSupplies[speciesIndex]++; // Increment the number of supplies for the species
}

// Update a supply for a species
void updateSupply(int speciesIndex, int supplyIndex, const char* newSupply) {
   if (supplyIndex < 0 || supplyIndex >= numSupplies[speciesIndex]) {
      printf("Invalid supply index.\n");
      return;
   }
   // Free the old supply and allocate memory for the new one
   free(speciesSupplies[speciesIndex][supplyIndex]);
   speciesSupplies[speciesIndex][supplyIndex] = (string)malloc(strlen(newSupply) + 1);
   strcpy(speciesSupplies[speciesIndex][supplyIndex], newSupply);
}
```

```
// Remove a species from the inventory
void removeSpecies(int speciesIndex) {
    // Free the supplies of the species
    for (int i = 0; i < numSupplies[speciesIndex]; i++) {
        free(speciesSupplies[speciesIndex][i]);
    }
    free(speciesSupplies[speciesIndex]); // Free the species' supply list

    // Shift other species' supplies to fill the gap
    for (int i = speciesIndex; i < numSpecies - 1; i++) {
        speciesSupplies[i] = speciesSupplies[i + 1];
        numSupplies[i] = numSupplies[i + 1];
    }

    // Reduce the number of species
    numSpecies--;
}

// Display inventory in 2D format
void displayInventory() {
    printf("\nInventory:\n");

    // Print the header with species names
    for (int i = 0; i < numSpecies; i++) {
        printf("%-15s", columnsName[i]);
    }
    printf("\n");

    // Print supplies for each species
    for (int i = 0; i < numSpecies; i++) {
        for (int j = 0; j < numSupplies[i]; j++) {
            if (j == 0) {
                printf("%-15s", columnsName[i]); // Print species name once per row
            }
            printf("%-15s", speciesSupplies[i][j]); // Print each supply
        }
        printf("\n");
    }
}

// Free allocated memory
void freeInventory() {
    for (int i = 0; i < numSpecies; i++) {
        for (int j = 0; j < numSupplies[i]; j++) {
            free(speciesSupplies[i][j]); // Free each supply
        }
        free(speciesSupplies[i]); // Free the species' supply list
    }
    free(speciesSupplies); // Free the 2D array of species supplies
    free(numSupplies); // Free the array tracking the number of supplies
}

int main() {
    // Initialize the inventory
    initializeInventory();

    // Adding supplies for "Dogs"
    addSupply(0, "Food");
    addSupply(0, "Leash");
    addSupply(0, "Toys");

    // Adding supplies for "Cats"
    addSupply(1, "Food");
    addSupply(1, "Litter");

    // Adding supplies for "Parrots"
    addSupply(2, "Seed");
    addSupply(2, "Cage");

    // Displaying inventory before updating
    printf("Before updating:\n");
    displayInventory();

    // Updating a supply for "Dogs" (Leash -> Collar)
    updateSupply(0, 1, "Collar");

    // Displaying inventory after update
    printf("\nAfter updating:\n");
    displayInventory();

    // Removing "Cats" species
    removeSpecies(1);

    // Displaying inventory after removal
    printf("\nAfter removing Cats:\n");
    displayInventory();
    // Free allocated memory
    freeInventory();

    return 0;
}
```

Reference for Q5 solution:
https://stackoverflow.com/questions/40568045/how-to-allocate-contiguous-2d-array-of-strings-in-c

Question 06

In a cutting-edge agritech system for precision farming, a dynamic pointer-based architecture is deployed to seamlessly manage interconnected data across fields, crops, weather, and smart equipment. Each **field** is represented as a structure containing GPS coordinates, soil health metrics, and moisture levels, alongside a pointer to a dynamically allocated array of **crop structures**. These **crop structures** store critical details like crop type, growth stage, and expected yield while maintaining pointers to **weather forecast structures** that provide hyper-local predictions for temperature, rainfall, and wind patterns.

The **field structure** also includes a pointer to an **equipment array**, representing key farming tools like autonomous tractors, irrigation systems, and drones. Each equipment structure tracks operational status, fuel levels, and activity schedules, enabling synchronized field operations. In addition, an array of **sensor structures** is linked to each field, capturing real-time data on soil nutrients, pH levels, and pest activity, empowering farmers to make informed decisions.

To scale operations, fields are grouped into **regional hubs**, each represented by a structure with pointers to arrays of fields. These hubs maintain aggregate data like yield predictions, resource distribution, and emergency response plans. All regional hubs are connected to a central analytics server through pointers, allowing AI algorithms to process massive datasets and generate real-time insights on crop health, irrigation efficiency, and equipment optimization.

This system's dynamic design ensures that every byte of memory is utilized efficiently, enabling rapid scaling and adaptation to environmental conditions. By leveraging advanced pointer structures, the agritech platform offers farmers a futuristic, data-driven farming experience that maximizes yield, minimizes waste, and supports sustainable agricultural practices.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structure definitions
struct WeatherForecast {
    float temperature;
    float rainfall;
    float wind_speed;
};
struct Crop {
    char crop_type[50];
    int growth_stage;
    float expected_yield;
    struct WeatherForecast* weather_forecast;
};
struct Field {
    float gps_latitude;
    float gps_longitude;
    float soil_health;
    float moisture_level;
    struct Crop* crops;
    int crop_count;
};
struct RegionalHub {
    struct Field* fields;
    int field_count;
};
struct CentralServer {
    struct RegionalHub** hubs;
    int hub_count;
};
// Function prototypes
struct WeatherForecast* create_weather_forecast(float temperature, float rainfall, float
wind_speed);
struct Crop* create_crop(const char* crop_type, int growth_stage, float expected_yield, struct
WeatherForecast* forecast);
struct Field* create_field(float latitude, float longitude, float soil_health, float moisture_level, int
crop_count);
struct RegionalHub* create_hub(int field_count);
void connect_hubs_to_server(struct CentralServer* server, struct RegionalHub** hubs, int
hub_count);
void cleanup(struct CentralServer* server);
int main() {
    // Initialize central server
    struct CentralServer central_server = {0, NULL};

    // Create weather forecast
    struct WeatherForecast* wf = create_weather_forecast(25.5, 10.2, 5.0);

    // Create crops
    struct Crop* crop = create_crop("Wheat", 2, 150.0, wf);

    // Create a field and assign crops
    struct Field* field = create_field(25.0, 71.5, 85.0, 60.0, 1);
    field->crops[0] = *crop;

    // Create a regional hub and assign fields
    struct RegionalHub* hub = create_hub(1);
    hub->fields[0] = *field;

    // Add hub to central server
    struct RegionalHub* hubs[] = {hub};
    connect_hubs_to_server(&central_server, hubs, 1);

    printf("System initialized with %d hub(s).\n", central_server.hub_count);

    // Cleanup allocated memory
    cleanup(&central_server);

    return 0;
}

// Function implementations
struct WeatherForecast* create_weather_forecast(float temperature, float rainfall, float
wind_speed) {
    struct WeatherForecast* wf = (struct WeatherForecast*)malloc(sizeof(struct
WeatherForecast));
    wf->temperature = temperature;
    wf->rainfall = rainfall;
    wf->wind_speed = wind_speed;
    return wf;
}

struct Crop* create_crop(const char* crop_type, int growth_stage, float expected_yield, struct
WeatherForecast* forecast) {
    struct Crop* crop = (struct Crop*)malloc(sizeof(struct Crop));
    strcpy(crop->crop_type, crop_type);
    crop->growth_stage = growth_stage;
    crop->expected_yield = expected_yield;
    crop->weather_forecast = forecast;
    return crop;
}
```

```
struct Field* create_field(float latitude, float longitude, float soil_health, float moisture_level, int
crop_count) {
    struct Field* field = (struct Field*)malloc(sizeof(struct Field));
    field->gps_latitude = latitude;
    field->gps_longitude = longitude;
    field->soil_health = soil_health;
    field->moisture_level = moisture_level;
    field->crop_count = crop_count;
    field->crops = (struct Crop*)malloc(sizeof(struct Crop) * crop_count);
    return field;
}

struct RegionalHub* create_hub(int field_count) {
    struct RegionalHub* hub = (struct RegionalHub*)malloc(sizeof(struct RegionalHub));
    hub->field_count = field_count;
    hub->fields = (struct Field*)malloc(sizeof(struct Field) * field_count);
    return hub;
}

void connect_hubs_to_server(struct CentralServer* server, struct RegionalHub** hubs, int
hub_count) {
    server->hub_count = hub_count;
    server->hubs = (struct RegionalHub**)malloc(sizeof(struct RegionalHub*) * hub_count);
    for (int i = 0; i < hub_count; i++) {
        server->hubs[i] = hubs[i];
    }
}

void cleanup(struct CentralServer* server) {
    for (int i = 0; i < server->hub_count; i++) {
        struct RegionalHub* hub = server->hubs[i];
        for (int j = 0; j < hub->field_count; j++) {
            struct Field* field = &hub->fields[j];
            free(field->crops);
        }
        free(hub->fields);
        free(hub);
    }
    free(server->hubs);
}
```

Question 07

In a Netflix-like streaming platform, 2D pointers are used to dynamically manage and personalize the viewing experience for millions of users across diverse content categories and device types. The platform employs a 2D pointer structure, where each row represents a **user profile**, and each column corresponds to a **content category** (e.g., Action, Drama, Comedy). A double** engagementMatrix pointer points to this 2D array, where each element stores a numerical engagement score (e.g., average viewing time or like/dislike ratio) for a user's interaction with that category.

Each user profile structure includes a pointer to their respective row in the engagement matrix, allowing for quick retrieval and updates of personalized data. For example, engagementMatrix[userIndex][categoryIndex] can be updated whenever the user streams content from a specific category, dynamically recalibrating their preferences in real-time.

The system also uses a secondary 2D pointer structure to manage **device-specific preferences**. For instance, deviceMatrix[userIndex][deviceIndex] points to dynamically allocated arrays holding resolution preferences, playback history, and bandwidth usage for different devices (smart TVs, laptops, smartphones) associated with a user's profile. This enables seamless transitions between devices while maintaining personalized settings like resolution and playback position.

Additionally, another 2D pointer system tracks **content metadata**, where each row corresponds to a content category and each column represents a specific piece of content. Each element in this matrix contains a pointer to a structure with attributes like title, rating, runtime, and encoding formats, enabling quick access to metadata for streaming.

This multi-layered 2D pointer-based design allows the platform to efficiently store, retrieve, and update personalized recommendations, device preferences, and content metadata. By leveraging such dynamic data structures, the system delivers a highly tailored, device-optimized viewing experience for users, ensuring maximum engagement and satisfaction while handling the scalability needs of a global user base.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structure for Content Metadata
struct ContentMetadata {
   char title[100];
   float rating;
   int runtime; // in minutes
   char encoding_format[20];
};
```

```
// Function prototypes
double** createEngagementMatrix(int users, int categories);
char*** createDevicePreferences(int users, int devices);
struct ContentMetadata*** createContentMatrix(int categories, int contents);
void freeEngagementMatrix(double** matrix, int users);
void freeDevicePreferences(char*** matrix, int users, int devices);
void freeContentMatrix(struct ContentMetadata*** matrix, int categories, int contents);

int main() {
    int numUsers = 3;
    int numCategories = 3;
    int numDevices = 2;
    int numContents = 3;

    // Create engagement matrix
    double** engagementMatrix = createEngagementMatrix(numUsers, numCategories);

    // Create device preferences
    char*** deviceMatrix = createDevicePreferences(numUsers, numDevices);

    // Create content metadata matrix
    struct ContentMetadata*** contentMatrix = createContentMatrix(numCategories,
numContents);

    // Example usage: Update engagement score
    engagementMatrix[0][1] = 4.5; // User 0, Category 1
    printf("User 0, Category 1 Engagement Score: %.1f\n", engagementMatrix[0][1]);

    // Example usage: Add a device preference
    strcpy(deviceMatrix[0][1], "1080p"); // User 0, Device 1
    printf("User 0, Device 1 Resolution: %s\n", deviceMatrix[0][1]);

    // Example usage: Add content metadata
    strcpy(contentMatrix[0][0]->title, "Action Movie 1");
    contentMatrix[0][0]->rating = 8.5;
    contentMatrix[0][0]->runtime = 120;
    strcpy(contentMatrix[0][0]->encoding_format, "H.264");
    printf("Content: %s, Rating: %.1f, Runtime: %d min, Format: %s\n",
           contentMatrix[0][0]->title,
           contentMatrix[0][0]->rating,
           contentMatrix[0][0]->runtime,
           contentMatrix[0][0]->encoding_format);
    // Free all dynamically allocated memory
    freeEngagementMatrix(engagementMatrix, numUsers);
    freeDevicePreferences(deviceMatrix, numUsers, numDevices);
    freeContentMatrix(contentMatrix, numCategories, numContents);

    return 0;
}

// Function to create engagement matrix
double** createEngagementMatrix(int users, int categories) {
    double** matrix = (double**)malloc(users * sizeof(double*));
    for (int i = 0; i < users; i++) {
        matrix[i] = (double*)malloc(categories * sizeof(double));
        for (int j = 0; j < categories; j++) {
            matrix[i][j] = 0.0; // Initialize to 0
        }
    }
    return matrix;
}

// Function to create device preferences matrix
char*** createDevicePreferences(int users, int devices) {
    char*** matrix = (char***)malloc(users * sizeof(char**));
    for (int i = 0; i < users; i++) {
        matrix[i] = (char**)malloc(devices * sizeof(char*));
        for (int j = 0; j < devices; j++) {
            matrix[i][j] = (char*)malloc(50 * sizeof(char)); // Allocate for string
            strcpy(matrix[i][j], "Default"); // Initialize to "Default"
        }
    }
    return matrix;
}

// Function to create content metadata matrix
struct ContentMetadata*** createContentMatrix(int categories, int contents) {
    struct ContentMetadata*** matrix = (struct ContentMetadata***)malloc(categories *
sizeof(struct ContentMetadata**));
    for (int i = 0; i < categories; i++) {
        matrix[i] = (struct ContentMetadata**)malloc(contents * sizeof(struct ContentMetadata*));
        for (int j = 0; j < contents; j++) {
            matrix[i][j] = (struct ContentMetadata*)malloc(sizeof(struct ContentMetadata));
            strcpy(matrix[i][j]->title, "");
            matrix[i][j]->rating = 0.0;
            matrix[i][j]->runtime = 0;
            strcpy(matrix[i][j]->encoding_format, "");
        }
    }
    return matrix;
}

// Function to free engagement matrix
void freeEngagementMatrix(double** matrix, int users) {
    for (int i = 0; i < users; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

// Function to free device preferences matrix
void freeDevicePreferences(char*** matrix, int users, int devices) {
    for (int i = 0; i < users; i++) {
        for (int j = 0; j < devices; j++) {
            free(matrix[i][j]);
        }
        free(matrix[i]);
    }
    free(matrix);
}

// Function to free content metadata matrix
void freeContentMatrix(struct ContentMetadata*** matrix, int categories, int contents) {
    for (int i = 0; i < categories; i++) {
        for (int j = 0; j < contents; j++) {
            free(matrix[i][j]);
        }
        free(matrix[i]);
    }
    free(matrix);
}
```