SNAKEWARE

THE SNAKE AND FOOD GAME

ALGORITHM

PART 1: INITIALIZATION

1. **Set Terminal to Non-blocking Input Mode:**
    - Configure the terminal to allow real-time input without waiting for the Enter key.
    - Disable echoing of input characters to prevent cluttering the display.
2. **Load High Scores:**
    - Read the `scores.txt` file to load previous high scores into an array of `PlayerScore`.
    - Initialize the number of players (`playerCount`) based on the file's content.
3. **Get Player Name:**
    - Prompt the player to enter their name.
    - Check if the name already exists in the high scores.
    - If the name exists:
        - Ask if the player wants to reuse it.
        - If not, prompt for a different name.
    - If the name does not exist, proceed with the entered name.
4. **Initialize Game Variables:**
    - Set the initial position of the snake at the center of the grid.
    - Set the snake's direction to `RIGHT`.
    - Initialize the snake's length to 1.
    - Place the first food item at a random position within the grid bounds.
    - Set the initial score to 0 and the speed to a predefined value (`INITIAL_SPEED`).

PART 2: GAMEPLAY

Display the Game Grid:

- Clear the screen.
- Show the current player's name and score.
- Display the highest score and the corresponding player.
- Print the game grid:
    - Represent the snake using the character `O`.
    - Represent the food using the character `F`.
    - Represent empty cells with the character.

Handle User Input:

- Check if a key has been pressed:
  - If `p` is pressed, pause the game and wait for any key to resume.
  - If an arrow key or `W`, `A`, `S`, `D` is pressed, change the snake's direction.
  - Ignore invalid input or directions that reverse the current direction.
  -

Move the Snake:

- Shift each segment of the snake to the position of the segment ahead of it.
- Update the snake's head based on the current direction (`UP`, `DOWN`, `LEFT`, or `RIGHT`).

Check Collisions:

- **Wall Collision:** If the snake's head moves out of bounds, end the game.
- **Self-Collision:** If the snake's head overlaps with any part of its body, end the game.

Handle Food Consumption:

- If the snake's head position matches the food's position:
  - Increase the snake's length by one.
  - Increment the score.
  - Generate a new food position at random, ensuring it does not overlap with the snake.
  - Increase the game speed by decreasing the delay (`speed`), ensuring it doesn't go below a minimum value.

PART 3: END GAME

1. **Display Game Over Message:**
   - Show the player's final score.
2. **Update High Scores:**
   - Check if the player's score is higher than their previous score (if they exist in the high scores).
   - If so, update the player's score in the `PlayerScore` array.
   - If the player is new, add their name and score to the array.
3. **Save High Scores to File:**
   - Open `scores.txt` in write mode.
   - Write all players' names and scores back to the file.
4. **Exit the Game:**
   - Terminate the program.

LOGIC CHECK (DRY RUN)

1. MOVEMENT

ASSUMPTIONS:

1. **Initial grid dimensions**: 5x5.
2. **Initial snake position**: Center of the grid `(2, 2)`.
3. **Initial food position**: Randomly generated, e.g., `(0, 0)`.
4. **Player Name**: `"Alice"`.
5. **Initial direction**: `RIGHT`.
6. **Snake length**: `1` (single segment).
7. **Input sequence**: `RIGHT`, `RIGHT`, `DOWN`, `LEFT`.

PART 1: INITIALIZATION

- **Snake Head Position**: `(2, 2)`.
- **Snake Body**: `[(2, 2)]` (only one segment).
- **Food Position**: `(0, 0)`.
- **Score**: `0`.
- **Speed**: `200000`.

STEP 2: GAMEPLAY

Initial grid:

```
F . . . .
. . . . .
. . O . .
. . . . .
. . . . .
```

Iteration 1: **Move Snake to the RIGHT**

User Input: `RIGHT`

1. **Update Snake Position**:
   - Move the head from `(2, 2)` to `(3, 2)`.
   - Update the snake body: `[(3, 2)]`.
2. **Collision Check**:
   - No wall collision (still inside grid).
   - No self-collision (only one segment).
3. **Food Check**:
   - Snake did not reach food (`(3, 2) != (0, 0)`).
4. **Grid Display**:

```
F . . . .
. . . . .
. . . O .
. . . . .
. . . . .
```

5. **Score:** 0

Iteration 2: **Move Snake to the RIGHT**

User Input: `RIGHT`

1. **Update Snake Position**:
   - Move the head from `(3, 2)` to `(4, 2)`.
   - Update the snake body: `[(4, 2)]`.
2. **Collision Check**:
   - No wall collision.
   - No self-collision.
3. **Food Check**:
   - Snake did not reach food (`(4, 2) != (0, 0)`).
4. **Grid Display**:

```
F . . . .
. . . . .
. . . . O
. . . . .
. . . . .
```

5. **Score**: `0`.

Iteration 3: **Move Snake DOWN**

User Input: `DOWN`

1. **Update Snake Position**:
   - Move the head from `(4, 2)` to `(4, 3)`.
   - Update the snake body: `[(4, 3)]`.
2. **Collision Check**:
   - No wall collision.
   - No self-collision.
3. **Food Check**:
   - Snake did not reach food (`(4, 3) != (0, 0)`).
4. **Grid Display**:

```
F . . . .
. . . . .
. . . . .
. . . . O
. . . . .
```

5. **Score**: `0`.

Iteration 4: **Move Snake LEFT**

User Input: `LEFT`

1. **Update Snake Position**:

    - Move the head from `(4, 3)` to `(3, 3)`.
    - Update the snake body: `[(3, 3)]`.
2. **Collision Check**:
    - No wall collision.
    - No self-collision.
3. **Food Check**:
    - Snake did not reach food (`(3, 3) != (0, 0)`).
4. **Grid Display**:

```
F . . . .
. . . . .
. . . . .
. . . O .
. . . . .
```

5. **Score**: `0`.

Iteration 5: **Food Consumption**

User Input: `UP` (Snake goes toward `(0, 0)` where food is placed).

1. **Update Snake Position**:
    - Move head to `(0, 0)` (where food is located).
    - Grow snake:
        - New length: `2`.
        - New body: `[(0, 0), (3, 3)]`.
2. **Collision Check**:
    - No wall collision.
    - No self-collision.
3. **Food Check**:
    - Food eaten! Update food to a new random position (e.g., `(4, 4)`).
4. **Grid Display**:

```
O O . . .
. . . . .
. . . . .
. . . . .
. . . . F
```

5. **Score**: `1`.

Endgame.

2. COLLISION

T**wo types of collisions**:

1. **Wall Collision**: The snake hits a boundary of the grid.
2. **Self-Collision**: The snake moves into its own body.

Collision Scenario 1: Wall Collision

Setup:

1. Grid size: **5x5**.
2. Initial snake position: Head at `(2, 4)` (near the top-right corner).
3. Direction: **RIGHT**.
4. Input sequence: `RIGHT`.

Dry Run:

1. **Initial Grid**:

```
. . . . .
. . . . .
. . . . O
. . . . .
. . . . .
```

2. **Snake Movement**:
   - The snake's head moves one step to the **right**: `(2, 5)`.
   - The new position is **outside the grid**.
3. **Collision Check**:
   - Check if `snake[0].x < 0`, `snake[0].x >= WIDTH`, `snake[0].y < 0`, or `snake[0].y >= HEIGHT`.
   - Condition `snake[0].x == 5` is **true** (out of bounds).
4. **Result**:
   - The game detects a wall collision.
   - Message displayed: "**Game Over! Final Score: X**".
   - The game exits, and the score is saved.

Collision Scenario 2: Self-Collision

Setup:

1. Grid size: **5x5**.
2. Snake position:
   - Head: `(2, 2)`.
   - Body: `[(2, 2), (2, 3), (3, 3), (3, 2)]` (snake forms a loop).
3. Direction: **UP**.
4. Input sequence: `LEFT`.

Dry Run:

1. **Initial Grid**:

```
. . . . .
. . . O .
. . . O O
. . O . .
. . . . .
```

2. **Snake Movement**:
   - Move head to `(1, 2)` (left of the current head).
   - Update the snake body:
     - Old tail `(3, 2)` is removed.
     - New body: `[(1, 2), (2, 2), (2, 3), (3, 3)]`.
3. **Collision Check**:
   - Loop through all segments of the body (`for i in 1 to snakeLength - 1`).
   - Check if `snake[0].x == snake[i].x` and `snake[0].y == snake[i].y`.
   - Condition `snake[0].x == 2 && snake[0].y == 2` is **true** (head collides with its own body).
4. **Result**:
   - The game detects a self-collision.
   - Message displayed: "**Game Over! Final Score: X**".
   - The game exits, and the score is saved.

LINE BY LINE DECODING

Header Inclusions

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  // For usleep
#include <termios.h> // For terminal input handling
#include <fcntl.h>   // For non-blocking input
#include <time.h>    // For random number generation
#include <ctype.h>   // For tolower function
#include <string.h>  // For string handling
```

stdio.h: Standard I/O library for input and output functions like `printf`, `scanf`.
stdlib.h: Provides functions like `exit`, `rand` for random number generation, and memory allocation.
unistd.h: Provides `usleep`, which is used for adding delays (in microseconds).
termios.h: Provides functions to manipulate terminal behavior (e.g., to disable echoing input).
fcntl.h: Allows setting non-blocking input mode on the terminal.
time.h: Provides functions related to time, particularly useful for seeding the random number generator.
ctype.h: Provides functions like `tolower` to handle character case.
string.h: Provides string manipulation functions like `strcmp`, `strcpy`.

Macro Definitions

```c
#define WIDTH 30
#define HEIGHT 30
#define INITIAL_SPEED 200000
#define SCORE_FILE "scores.txt"
```

- **WIDTH** and **HEIGHT** define the dimensions of the grid on which the snake game is played.
- **INITIAL_SPEED** defines the initial delay between each frame of the game (controls the snake's speed).
- **SCORE_FILE** is the name of the file where the high scores are stored.

Enums and Structs

```
// Directions
enum { UP, DOWN, LEFT, RIGHT };
```

This defines an enum for the four possible directions (`UP = 0, DOWN = 1, LEFT = 2, RIGHT = 3`).

```
typedef struct
{
    int x, y;
} Position;
```

Defines a `Position` structure to hold the `x` and `y` coordinates of a point in the game (e.g., snake's body or food).

```
typedef struct
{
    char name[50];
    int score;
} PlayerScore;
```

Defines a `PlayerScore` structure to store a player's name and score.

```
PlayerScore playerScores[100];
int playerCount = 0;
```

playerScores is an array of `PlayerScore` to store up to 100 players' names and their scores.
playerCount keeps track of how many players' scores are stored.

```
Position snake[900]; // Max length of the snake
int snakeLength;
Position food;
int direction;
int score;
int speed; // Speed of the game
char playerName[50];
```

snake: Array to hold the positions of the snake's body (maximum length of 900).
snakeLength: The current length of the snake.
food: The position of the food on the grid.
direction: The current direction of the snake (one of UP, DOWN, LEFT, RIGHT).
score: The player's current score.
speed: The delay time between each frame (controls the speed of the snake).
playerName: A string to store the name of the player.

Non-blocking Input Functions

```
void setNonBlockingInput()
{
    struct termios t;
    tcgetattr(STDIN_FILENO, &t);
    t.c_lflag &= ~ICANON; // Disable canonical mode
    t.c_lflag &= ~ECHO;   // Disable echo
    tcsetattr(STDIN_FILENO, TCSANOW, &t);
}
```

Disables canonical mode (buffered input) and echoing of typed characters, making input immediately available without waiting for the Enter key.
It uses the **termios** API to change terminal settings.

```c
int kbhit()
{
    struct termios oldt, newt;
    int ch;
    int oldf;

    tcgetattr(STDIN_FILENO, &oldt);
    newt = oldt;
    newt.c_lflag &= ~(ICANON | ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
    oldf = fcntl(STDIN_FILENO, F_GETFL, 0);
    fcntl(STDIN_FILENO, F_SETFL, oldf | O_NONBLOCK);

    ch = getchar();

    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    fcntl(STDIN_FILENO, F_SETFL, oldf);

    if (ch != EOF)
    {
        ungetc(ch, stdin);
        return 1;
    }

    return 0;
}
```

- Checks if a key has been pressed without blocking the program. Uses **termios** and **fcntl** to enable non-blocking mode.
- Returns `1` if a key was pressed, otherwise `0`.

```c
char getch()
{
    return getchar();
}
```

A function to read a single character from the keyboard. This is used to capture player input.

Game Score Management

```
void loadScores()
{
    FILE *file = fopen(SCORE_FILE, "r");
    if (file == NULL)
        return;

    playerCount = 0;
    while (fscanf(file, "%s %d", playerScores[playerCount].name, &playerScores[
playerCount].score) != EOF)
    {
        playerCount++;
    }

    fclose(file);
}
```

- Loads the high scores from the file `scores.txt` into the **playerScores** array.
- The function reads the player name and score in pairs (`name score`) until the end of the file.

```
void updateScoreFile()
{
    int found = 0;

    // Update score if player exists
    for (int i = 0; i < playerCount; i++)
    {
        if (strcmp(playerScores[i].name, playerName) == 0)
        {
            if (score > playerScores[i].score)
            {
                playerScores[i].score = score; // Update only if the score is higher
            }
            found = 1;
            break;
        }
    }

    // Add new player if not found
    if (!found)
    {
        strcpy(playerScores[playerCount].name, playerName);
        playerScores[playerCount].score = score;
        playerCount++;
    }

    // Save updated scores
    FILE *file = fopen(SCORE_FILE, "w");
    for (int i = 0; i < playerCount; i++)
    {
        fprintf(file, "%s %d\n", playerScores[i].name, playerScores[i].score);
    }

    fclose(file);
}
```

Updates the high scores in the `scores.txt` file. If the current player's score is higher than the saved score, it updates it; if the player doesn't exist, it adds them.

Game Initialization

```
void initializeGame()
{
    snakeLength = 1;
    snake[0].x = WIDTH / 2;
    snake[0].y = HEIGHT / 2;
    direction = RIGHT;
    score = 0;
    speed = INITIAL_SPEED; // Start with the initial speed

    srand(time(0));
    food.x = rand() % WIDTH;
    food.y = rand() % HEIGHT;

    loadScores();
}
```

Initializes the game state: sets the starting position and length of the snake, initial score, and food position. **srand(time(0))** is used to seed the random number generator to ensure random food positions.

Checking If the Snake Occupies a Position

```
int isSnake(int x, int y)
{
    for (int i = 0; i < snakeLength; i++)
    {
        if (snake[i].x == x && snake[i].y == y)
            return 1;
    }
    return 0;
}
```

Checks if a given position `(x, y)` is occupied by the snake body.

Drawing the Grid

```c
void drawGrid()
{
    system("clear");
    printf("Player: %s | Score: %d\n", playerName, score);

    int maxScore = 0;
    char maxPlayer[50];
    for (int i = 0; i < playerCount; i++)
    {
        if (playerScores[i].score > maxScore)
        {
            maxScore = playerScores[i].score;
            strcpy(maxPlayer, playerScores[i].name);
        }
    }
    printf("Highest Score: %s (%d)\n", maxPlayer, maxScore);

    for (int i = 0; i < HEIGHT; i++)
    {
        for (int j = 0; j < WIDTH; j++)
        {
            if (isSnake(j, i))
            {
                printf("O ");
            }
            else if (food.x == j && food.y == i)
            {
                printf("F ");
            }
            else
            {
                printf(". ");
            }
        }
        printf("\n");
    }
}
```

Clears the screen and prints the current state of the game grid, showing the snake (`O`), food (`F`), and empty spaces (`.`). Printing the current score along with the player's name and also the highest scoring player and the score.

Updating the Snake Body

```c
void updateSnakeBody()
{
    for (int i = snakeLength - 1; i > 0; i--)
    {
        snake[i] = snake[i - 1];
    }
}
```

This function shifts the body parts of the snake one position forward in the snake's body array. The last segment of the snake follows the second-to-last segment, and so on, allowing the snake to "move" forward.

Updating the Snake Head

```c
void updateSnakeHead()
{
    switch (direction)
    {
    case UP:
        snake[0].y--;
        break;
    case DOWN:
        snake[0].y++;
        break;
    case LEFT:
        snake[0].x--;
        break;
    case RIGHT:
        snake[0].x++;
        break;
    }
}
```

Updates the position of the snake's head based on the current direction (up, down, left, right).
The head's position is updated first, before checking collisions or moving the body.

Collission Detection

```c
void checkCollisions()
{
    if (snake[0].x < 0 || snake[0].x >= WIDTH || snake[0].y < 0 || snake[0].y >= HEIGHT)
    {
        printf("Game Over! Final Score: %d\n", score);
        updateScoreFile();
        exit(0);
    }

    for (int i = 1; i < snakeLength; i++)
    {
        if (snake[0].x == snake[i].x && snake[0].y == snake[i].y)
        {
            printf("Game Over! Final Score: %d\n", score);
            updateScoreFile();
            exit(0);
        }
    }
}
```

Ends the game if the snake collides with the wall or itself.
Displays the final score and updates the score file.

Increasing Speed

```
void increaseSpeed()
{
    if (snakeLength % 5 == 0)
    {
        speed = speed > 50000 ? speed - 2000 : speed;
    }
}
```

Increases the game speed every 5 snake length increments by reducing the delay time.

Moving the Snake

```
void moveSnake()
{
    updateSnakeBody();
    updateSnakeHead();
    checkCollisions();

    if (snake[0].x == food.x && snake[0].y == food.y)
    {
        snakeLength++;
        score++;
        food.x = rand() % WIDTH;
        food.y = rand() % HEIGHT;
        increaseSpeed();
    }
}
```

Moves the snake by updating its body and head positions.
Checks for collisions.
If the snake eats the food, it grows, the score increases, and new food is randomly placed.

Changing Snake Direction

```
void changeDirection(char key)
{
    if (key == 27)
    {
        char secondKey = getchar();
        if (secondKey == 91)
        {
            char arrowKey = getchar();
            switch (arrowKey)
            {
            case 'A':
                if (direction != DOWN)
                    direction = UP;
                break;
            case 'B':
                if (direction != UP)
                    direction = DOWN;
                break;
            case 'C':
                if (direction != LEFT)
                    direction = RIGHT;
                break;
            case 'D':
                if (direction != RIGHT)
                    direction = LEFT;
                break;
            }
        }
    }
    else
    {
        switch (tolower(key))
        {
        case 'w':
            if (direction != DOWN)
                direction = UP;
            break;
        case 's':
            if (direction != UP)
                direction = DOWN;
            break;
        case 'a':
            if (direction != RIGHT)
                direction = LEFT;
            break;
        case 'd':
            if (direction != LEFT)
                direction = RIGHT;
            break;
        }
    }
}
```

Changes the direction of the snake based on the player's keypress (`WASD` or arrow keys).

Main Function

```c
int main()
{
    setNonBlockingInput();
    initializeGame();

    int isNameValid;

    do
    {
        isNameValid = 1;
        printf("Enter your name:\n");
        scanf("%s", playerName);

        for (int i = 0; i < playerCount; i++)
        {
            if (strcmp(playerScores[i].name, playerName) == 0)
            {
                printf("Name '%s' already exists.\n
Do you want to use this name? (y/n):\n", playerName);
                char choice;
                scanf(" %c", &choice);

                if (tolower(choice) == 'y')
                {
                    printf("Welcome back, %s!\n", playerName);
                    isNameValid = 1;
                }
                else
                {
                    printf("Please enter a different name.\n");
                    isNameValid = 0;
                }
                break;
            }
        }
    } while (!isNameValid);

    printf("Welcome, %s!\n", playerName);
    printf("Press any key to start the game...\n");
    while (!kbhit())
        ;
    getch();

    while (1)
    {
        drawGrid();
        if (kbhit())
        {
            char key = getch();
            if (key == 'p')
            {
                printf("Game Paused. Press any key to continue...\n");
                while (!kbhit())
                    ;
                getch();
            }
            else
            {
                changeDirection(key);
            }
        }
        moveSnake();
        usleep(speed);
    }

    return 0;
}
```

Non-blocking Input Setup:

- `setNonBlockingInput()` ensures that the game can capture key presses without waiting for Enter to be pressed.

Initialize Game:

- `initializeGame()` sets up the initial state of the snake, food, score, and loads high scores.

Name Validation:

- The player is prompted to enter their name. If the name already exists in the score file, they are asked if they want to use it. This loop repeats until a valid name is entered.

Starting the Game:

- After the name is validated, a welcome message is displayed, and the game waits for the player to press any key to start.

Main Game Loop:

- The game continuously updates by:
  - Drawing the game grid.
  - Handling key presses for snake movement or pausing.
  - Moving the snake and checking for collisions.
  - Adjusting speed based on snake growth.

Game Continues:

- The game runs in a loop until the player loses (via collision) or quits the game.