



Softwarica

College of IT & E-commerce



Programming

Name: Abhishek Adhikari

Class: Foundation Group 37A

Email: adhikaryabhishek209@gmail.com

Submitted to Shrawan Thakur



Table of Contents

Smart Farm Management System Documentation	1
Project Overview.....	1
System Requirements or Installation	1
Running in Docker Container:	1
Running Locally:	1
Data Structures.....	2
Global Variables.....	2
File Constants	3
Main Functionalities	3
1. Crop Management Function.....	4
1.1. addCrop():	4
1.2. viewAllCrops():	5
1.3. updateCropStatus():	6
1.4. deleteCrop():.....	7
2. Irrigation Scheduling Function.....	7
2.1. inputFieldData():	8
2.2. calculateIrrigationNeed():.....	9
2.3. calculateWaterRequirement():	9
2.4. calculateWaterRequirement():	10
3. Expense Management Function.....	11
3.1. addExpense():	12
3.2. viewMonthlyExpenses():.....	12
3.3. calculateTotalAndAverageExpenses():	13
3.4. viewExpenseLog():	14
4. Exit and Loading (Utility) Functions	15
4.1. loadData():.....	15
4.2. saveData():	15
4.3 lowerCase(char *s).....	16
Memory Management.....	16
Error Handling	17

Terminal Formatting	17
Future Improvements.....	18
General Queries:	18
Why sscanf() instead of fscanf() ?.....	18
Why to use dynamic memory allocation like realloc(), calloc(), malloc() and free()?	18
What's the advantage of error handling with perror()?	18
References.....	19

Smart Farm Management System Documentation

Project Overview

Project Title: Smart Farm Management System

Author: Abhishek Adhikari

Date: 15th September 2024

Description: This project is a Smart Farm Management System developed in C, providing functionalities such as managing crop information, scheduling irrigation, calculating water requirements, tracking expenses, and saving/loading data. The system includes a simple command-line interface with color-coded terminal output for better user experience.

Hosting: The project is hosted using Docker for portability and ease of deployment across multiple systems. The code is available for download and contributions on [Github](#).

System Requirements or Installation

Running in Docker Container: The application is hosted in a Docker container to allow easy deployment. If you prefer running the application in a Docker container you can pull and run the Docker image using the following commands:

```
//Pull the Docker Image  
docker pull darkabhi5/swt-smart-farm-management
```

```
//Run the Docker Container  
docker run -it darkabhi5/swt-smart-farm-management
```

Running Locally: If you prefer to run the application on your own system (without Docker), follow these steps: To compile and run the program, you will need a C compiler. I recommend using the [GCC compiler](#) for Windows or an equivalent compiler for other systems.

Steps to install:

1. Download and install the [GCC compiler](#) for your system.
2. Clone or download the repository:

```
git clone https://github.com/Abhishek-Adhikari-1/swt-smart-farm-management-system.git
```

3. Navigate to the project directory, compile the code and run:

```
gcc -o farm_manager main.c ; if($?) {.\farm_manager}
```



Data Structures

The program uses three key structures:

1. **Crops:** Represents a crop in the farm, including information about the crop's type, area, yield, planting date, harvest date, and its current status.

```
typedef struct {
    char name[50];           // Name of the crop (e.g., wheat, rice)
    float area;              // Area of the crop (in acres)
    float yield;             // Yield in kilograms or tons
    char plantingDate[11];   // Planting date (format: YYYY-MM-DD)
    char harvestDate[11];    // Harvest date (format: YYYY-MM-DD)
    char status[20];         // Status of the crop (e.g., "Planted", "Harvested")
} Crop;
```

2. **Field:** Represents a farmer's field and its condition, including the type of crop, the area of the field, and the current soil moisture level.

```
typedef struct {
    char cropType[50];       // The type of crop grown in this field (e.g., wheat, maize)
    float area;              // Area of the field (in acres)
    float soilMoisture;      // Soil moisture level as a percentage (0-100%)
} Field;
```

3. **Expense:** Represents expenses incurred on the farm, with fields for the category of the expense, a description of what was purchased or paid for, and the amount spent.

```
typedef struct {
    char category[20];       // Category of the expense (e.g., "Labor", "Fertilizer")
    char description[100];   // A short description of the expense
    float amount;           // Amount spent (in local currency)
} Expense;
```

Global Variables

Global variables in the program help manage the dynamically allocated arrays for crops, fields, and expenses, as well as the total number of items in each array.

- **fields:** A dynamic array of type Field used to store information about each field in the system.
- **numFields:** An integer that tracks the number of fields currently stored in memory.
- **crops:** A dynamic array of type Crop used to store information about crops such as their type, area, and growth status.



- **numCrops**: An integer that keeps track of the number of crops entered in the system.
- **expenses**: A dynamic array of type Expense, storing farm-related expenses such as labor, machinery, or fertilizer.
- **numExpenses**: An integer tracking the total number of expenses recorded in the system.

? Why am I tracking number of crops, fields and expenses?

→ I am tracking them to allocate memory using realloc function to make the program memory efficient.

File Constants

The program uses predefined constant to handle file name and operation for storing and loading data:

- **Saving Data**: The saveData() function writes all current crops, and expenses into the file after section Crops: or Expenses: .
- **Loading Data**: The loadData() function reads from the file when the program starts, restoring any previously saved information.

#define FILENAME "farmerDetails.txt" is a predefined name of a file from which data should be load or save.

- **CROPS**: It is loaded from the same file defined in in `FILENAME` but have a format like Crops: and everything under it is a crop.
- **EXPENSES**: It is loaded from the same file defined in in `FILENAME` but have a format like Expenses: and everything under it is an expense.

Each data stores in a `FILENAME` is respective structure in a specific format, and these constants help manage file operations in a centralized way. There is no file data for irrigation because it contains soil moisture level and it should be verified each time.

Main Functionalities

The mainMenu() function serves as the primary interface for the **Smart Farm Management System**, allowing users to navigate between the major functionalities such as managing crops, scheduling irrigation, tracking expenses, and exiting the program. This menu acts as a central hub, directing users to different sections based on their choices. The system validates user input using scanf(). If the user enters something other than an integer (e.g., letters or special characters), it prints an error message.



This visual terminal shows the main-menu of the system.

```
Welcome to Smart Farm Management System
-----
1. Manage Crops
2. Irrigation Scheduling
3. Expense Tracking
4. Exit
Enter your choice: string
Invalid input. Please enter only an integer.
Enter your choice: |
```

1. Crop Management Function

The `manageCropsMenu()` function provides an interactive interface for users to manage the crops in the system. It offers multiple options such as adding, viewing, updating, and deleting crop details. The function also validates and gives warning in red text to the user. The function is implemented in a loop, ensuring that users can continue managing crops until they choose to return to the main menu. The menu offers the following options to the user:

```
Manage Crops Menu
-----
1. Add Crop Details
2. View All Crops
3. Update Crop Status
4. Delete Crop
5. Back to Main Menu
Enter your choice: char
Invalid input. Please enter only an integer.
Enter your choice: |
```

1.1. `addCrop()`:

- **Purpose:** Adds a new crop to the system, including details such as the crop name, area (in acres), expected yield, planting date, harvest date, and status.
- **Details:**
 - The function starts by reallocating memory for the crops array to store a new crop. It checks if memory allocation was successful, and if not, it returns an error.
 - Prompts the user to input crop details, ensuring validation for proper data formats.



- Dynamically allocates memory for each new crop added to the system using `realloc()`.
- Adds the new crop to the crops array and provides feedback on successful addition.
- **Visual:**

```

Manage Crops Menu
-----
1. Add Crop Details
2. View All Crops
3. Update Crop Status
4. Delete Crop
5. Back to Main Menu
Enter your choice: 1

Enter Crop Name: Sugarcane
Enter Area (in hectares): 13
Enter Yield (in tons): 23
Enter Planting Date (YYYY-MM-DD): 2020-08-16
Enter Harvest Date (YYYY-MM-DD): 2021-02-07
Enter Status (Planted | Harvested | Ready to Harvest): Harvested
Crop Added Successfully!
Press Enter to continue...

```

1.2. `viewAllCrops()`:

- **Purpose:** Displays a list of all crops currently recorded in the system, along with key details like area, yield, and status in a clear format like table.
- **Details:**
 - Outputs a table listing each crop's details:
 - Name
 - Area (in acres)
 - Yield (in tons)
 - Planting Date
 - Harvest Date
 - Status (e.g., "Planted", "Harvested", "Ready to Harvest")
 - If no crops have been added, it warns the user about the error.
 - Uses table formatting for clear and readable output in the terminal with colors.
- **Visual:**




```

Manage Crops Menu
-----
1. Add Crop Details
2. View All Crops
3. Update Crop Status
4. Delete Crop
5. Back to Main Menu
Enter your choice: 2

Crop Details:
+-----+-----+-----+-----+-----+-----+
| No. | Name | Area (hectares) | Yield (tons) | Planting Date | Harvest Date | Status |
+-----+-----+-----+-----+-----+-----+
| 1 | Rice | 22.00 | 657.00 | 2020-04-12 | 2020-08-26 | Harvested |
| 2 | Millet | 10.00 | 34.00 | 2022-02-15 | 2024-05-30 | Planted |
| 3 | Wheat | 50.00 | 125.00 | 2021-06-10 | 2022-09-15 | Ready_to_Harvest |
| 4 | Corn | 15.00 | 60.00 | 2023-03-25 | 2023-09-10 | Planted |
| 5 | Soybean | 30.00 | 85.00 | 2020-12-01 | 2021-08-20 | Harvested |
| 6 | Barley | 45.00 | 95.00 | 2021-10-05 | 2022-07-25 | Ready_to_Harvest |
| 7 | Oats | 25.00 | 70.00 | 2022-01-12 | 2022-06-18 | Harvested |
| 8 | Sunflower | 12.00 | 45.00 | 2023-04-05 | 2023-08-15 | Planted |
| 9 | Canola | 18.00 | 40.00 | 2022-11-20 | 2023-06-01 | Ready_to_Harvest |
| 10 | Peas | 8.00 | 20.00 | 2022-07-15 | 2023-03-10 | Harvested |
| 11 | Sugarcane | 13.00 | 23.00 | 2020-08-16 | 2021-02-07 | Harvested |
+-----+-----+-----+-----+-----+-----+
Press Enter to continue...

```

1.3. updateCropStatus():

- **Purpose:** Updates the status of a specific crop, allowing users to track changes in the crop's growth stage (e.g., from "Planted" to "Ready to Harvest" to "Harvested").
- **Details:**
 - Prompts the user to select a crop by its name and formats it on lowercase.
 - Asks for the new status of the crop showing current status for user access.
 - Validates that the input status is valid and matches predefined crop status options.
 - Updates the crop's status in the crops array and provides feedback on successful status update.
 - If no crops exist, the function warns the user saying no crops found in red color.
- **Visual:**

```

Manage Crops Menu
-----
1. Add Crop Details
2. View All Crops
3. Update Crop Status
4. Delete Crop
5. Back to Main Menu
Enter your choice: 3

Enter the name of the crop to update: rice
Updating status for crop 'rice':
Avoid using spaces. Instead use _
Enter new Status (Planted | Harvested | Ready_to_Harvest) [current: Harvested]: Planted
Crop Status Updated Successfully!
Press Enter to continue...

```



1.4. *deleteCrop()*:

- **Purpose:** Searches for and delete the crop details based on the crop name entered by the user.
- **Details:**
 - The function starts by reallocating memory for the crops array to delete a crop. It checks if memory allocation was successful, and if not, it returns an error.
 - Prompts the user to input the name of the crop they wish to delete.
 - If no crop with the specified name is found, it warns the user with red text.
 - If the crop is found, it removes the crop's details (name, area, yield, planting/harvest dates, status) and provides feedback on successful deletion.
- **Visual:**

```
Manage Crops Menu
-----
1. Add Crop Details
2. View All Crops
3. Update Crop Status
4. Delete Crop
5. Back to Main Menu
Enter your choice: 4

Enter the name of the crop to delete: sugarcane
Crop 'sugarcane' deleted successfully!
Press Enter to continue...|
```

2. Irrigation Scheduling Function

The `irrigationSchedulingMenu()` function provides the interface for managing irrigation-related tasks in the **Smart Farm Management System**. It allows users to input field data, calculate irrigation needs and water requirements, generate an irrigation schedule, and return to the main menu. The menu operates within a do-while loop to ensure that users can repeatedly perform actions until they choose to return to the main menu by selecting option 5. The user's input is validated to ensure it is an integer using `scanf()`. If the input is invalid (e.g., the user enters a non-integer value), an error message is displayed.



Irrigation Scheduling Menu

```
1. Input Field Data
2. Calculate Irrigation Need
3. Calculate Water Requirement
4. Generate Irrigation Schedule
5. Back to Main Menu
Enter your choice: |
```

2.1. *inputFieldData()*:

- **Purpose:** This function collects data about a field, including crop type, area, and soil moisture levels, and stores it dynamically for further use in irrigation calculations.
- **Details:**
 - The function starts by reallocating memory for the fields array to store a new field. It checks if memory allocation was successful, and if not, it returns an error.
 - A Field structure is created, which stores the field data: cropType, area, and soilMoisture.
 - Input validation:
 - For area, it ensures the input is a valid floating-point number.
 - For soilMoisture, it checks that the input is a valid percentage (between 0 and 100). If an invalid percentage is entered, the input is discarded and the user is asked to re-enter it.
 - After successful input, the new field is added to the fields array, and the field count is incremented.
 - A success message is displayed, confirming that the field data has been successfully added.
- **Visual:**

Irrigation Scheduling Menu

```
1. Input Field Data
2. Calculate Irrigation Need
3. Calculate Water Requirement
4. Generate Irrigation Schedule
5. Back to Main Menu
Enter your choice: 1
```

```
Enter Crop Type: Rice
Enter Area (in acres): 15
Enter Current Soil Moisture Level (percentage): 56
Field Data Added Successfully!
Press Enter to continue...|
```



2.2. *calculateIrrigationNeed()*:

- **Purpose:** This function determines whether each field requires irrigation based on its soil moisture levels.
- **Details:**
 - The function first checks if there are any fields available by checking fieldCount. If no fields have been added, it informs the user to input field data first.
 - It then iterates through all the fields and compares the soilMoisture to a threshold value (40%). If the soil moisture is below 40%, the field requires irrigation, and the function prints in yellow color those who needs irrigation and white who don't need.
 - If the soil moisture is equal to or greater than 40%, it prints that no irrigation is needed for that field.
- **Visual:**

```
Irrigation Scheduling Menu
-----
1. Input Field Data
2. Calculate Irrigation Need
3. Calculate Water Requirement
4. Generate Irrigation Schedule
5. Back to Main Menu
Enter your choice: 2

Field 1 (Crop: Rice) does not require irrigation.
Field 2 (Crop: Sugarcane) requires irrigation.
Field 3 (Crop: Millet) requires irrigation.
Field 4 (Crop: Wheat) does not require irrigation.
Field 5 (Crop: Corn) requires irrigation.
Field 6 (Crop: Soyabean) does not require irrigation.
Field 7 (Crop: Barley) does not require irrigation.
Field 8 (Crop: Oats) does not require irrigation.
Field 9 (Crop: Sunflower) does not require irrigation.
Field 10 (Crop: Peas) requires irrigation.
Press Enter to continue...|
```

2.3. *calculateWaterRequirement()*:

- **Purpose:** This function calculates the amount of water needed for each field based on its area and soil moisture level.
- **Details:**
 - Similar to the irrigation need function, it first checks if there are any fields added. If none exist, the user is prompted to input data.



- For each field, the formula used is:

```
Water needed = area * (100 - soilMoisture) * 10
```

- This formula calculates the water requirement in liters based on the area of the field and the difference between full soil moisture (100%) and the current soil moisture level.

- The result is printed for each field, specifying how much water (in liters) is needed.

- **Visual:**

```
Irrigation Scheduling Menu
-----
1. Input Field Data
2. Calculate Irrigation Need
3. Calculate Water Requirement
4. Generate Irrigation Schedule
5. Back to Main Menu
Enter your choice: 3

Field 1 (Crop: Rice) needed 6600.0 litres of water.
Field 2 (Crop: Sugarcane) needed 7680.0 litres of water.
Field 3 (Crop: Millet) needed 8800.0 litres of water.
Field 4 (Crop: Wheat) needed 29000.0 litres of water.
Field 5 (Crop: Corn) needed 9750.0 litres of water.
Field 6 (Crop: Soyabean) needed 13200.0 litres of water.
Field 7 (Crop: Barley) needed 14000.0 litres of water.
Field 8 (Crop: Oats) needed 8750.0 litres of water.
Field 9 (Crop: Sunflower) needed 1200.0 litres of water.
Field 10 (Crop: Peas) needed 6640.0 litres of water.
Press Enter to continue...|
```

2.4. *calculateWaterRequirement()*:

- **Purpose:** This function generates a detailed table showing the irrigation schedule for all fields, including crop type, water requirement, and whether irrigation is needed.
- **Details:**
 - Like the previous functions, it checks whether any fields have been added. If not, it prompts the user to input field data first.
 - The function then iterates over each field, calculates the water requirement using the same formula as `calculateWaterRequirement()`, and checks whether irrigation is needed.
 - A formatted table is printed to the terminal:
 - Field number: Identifies the field.
 - Crop: Shows the type of crop planted in the field.



- Water Needed: Displays the water requirement in liters for each field.
 - Irrigation Needed: Displays whether irrigation is required (Yes or No).
 - The irrigation schedule provides a comprehensive view of field irrigation needs in a tabular format.
- **Visual:**

```

Irrigation Scheduling Menu
-----
1. Input Field Data
2. Calculate Irrigation Need
3. Calculate Water Requirement
4. Generate Irrigation Schedule
5. Back to Main Menu
Enter your choice: 4
  
```

Field	Crop	Water Needed	Irrigation Needed
1	Rice	6600.0 ltr	No
2	Sugarcane	7680.0 ltr	Yes
3	Millet	8800.0 ltr	Yes
4	Wheat	29000.0 ltr	No
5	Corn	9750.0 ltr	Yes
6	Soyabean	13200.0 ltr	No
7	Barley	14000.0 ltr	No
8	Oats	8750.0 ltr	No
9	Sunflower	1200.0 ltr	No
10	Peas	6640.0 ltr	Yes

```

Press Enter to continue...
  
```

3. Expense Management Function

The `expenseTrackingMenu()` function is responsible for managing farm expenses within the Smart Farm Management System. This menu provides several options for users to add, view, and calculate expenses. The menu operates within a do-while loop to ensure that users can repeatedly perform actions until they choose to return to the main menu by selecting option 5. The user's input is validated to ensure it is an integer using `scanf()`. If the input is invalid (e.g., the user enters a non-integer value), an error message is displayed.

```

Expense Tracking Menu
-----
1. Add Expenses
2. View Monthly Expenses
3. Calculate Total and Average Expenses
4. View Expense Log
5. Back to Main Menu
Enter your choice: |
  
```



3.1. *addExpense()*:

- **Purpose:** This function allows the user to add a new expense entry to the system. It uses realloc to expand the expenses array dynamically, accommodating new entries.
- **Details:**
 - Prompts the user to enter the expense category and description. The user is advised to avoid spaces in the category and use underscores (_) instead.
 - Ensures the user enters a valid expense amount. If an invalid input is detected, the user is asked to re-enter the amount.
 - The new expense is added to the expenses array, and the total number of expenses (expenseCount) is updated.
 - After successfully adding the expense, a success message is displayed, and the system waits for the user to press any key to continue.
- **Visual:**

```
Expense Tracking Menu
-----
1. Add Expenses
2. View Monthly Expenses
3. Calculate Total and Average Expenses
4. View Expense Log
5. Back to Main Menu
Enter your choice: 1

Enter Expense Category: Cement
Avoid using spaces. Instead use _
Enter Expense Description: This_is_paid_to_babari
Enter Expense Amount: asd
Invalid input. Please enter a positive number for the amount.
Enter Expense Amount: 532
Expenses Added Successfully!
Press Enter to continue...|
```

3.2. *viewMonthlyExpenses()*:

- **Purpose:** Displays a formatted table of all the expenses added by the user.
- **Details:**
 - If no expenses are available, an error message is shown.
 - Otherwise, it prints a table with headings: **No.**, **Category**, **Amount**, and **Description**.



- Each row of the table represents an individual expense, displaying its category, amount, and description.
 - The system pauses after displaying the data, allowing the user time to review the expenses.
- **Visual:**

```
Expense Tracking Menu
-----
1. Add Expenses
2. View Monthly Expenses
3. Calculate Total and Average Expenses
4. View Expense Log
5. Back to Main Menu
Enter your choice: 2

Monthly Expenses:
+-----+-----+-----+-----+
| No. | Category | Amount | Description |
+-----+-----+-----+-----+
| 1 | labour | $56.00 | charge_of_temporary_labours |
| 2 | rod | $434.00 | fencing_rod |
| 3 | lunch | $6.00 | food_for_labours |
| 4 | fertilizer | $120.00 | crop_nutrients |
| 5 | seeds | $75.00 | various_seeds |
| 6 | pesticides | $90.00 | pest_control |
| 7 | water | $30.00 | irrigation |
| 8 | tools | $150.00 | farming_tools |
| 9 | transport | $200.00 | crop_transportation |
| 10 | maintenance | $45.00 | equipment_maintenance |
| 11 | Cement | $532.00 | This_is_paid_to_babari |
+-----+-----+-----+-----+
Press Enter to continue...|
```

3.3. *calculateTotalAndAverageExpenses()*:

- **Purpose:** This function calculates and displays the total and average amount of all expenses entered.
- **Details:**
 - If no expenses are recorded, it informs the user that no calculation can be done.
 - It loops through all expenses and sums up the amounts.
 - The average expense is calculated by dividing the total amount by the number of expenses.
 - Finally, it prints the total and average expense values to the user, and then pauses for review.



- **Visual:**

```
Expense Tracking Menu
-----
1. Add Expenses
2. View Monthly Expenses
3. Calculate Total and Average Expenses
4. View Expense Log
5. Back to Main Menu
Enter your choice: 3

Expense Summary:
Total Expenses: $ 1738.00
Average Expense: $ 158.00
Press Enter to continue...|
```

3.4. `viewExpenseLog()`:

- **Purpose:** This function displays a detailed log of all the expenses recorded.
- **Details:**
 - If there are no expenses, the user is informed of the lack of data.
 - Otherwise, the system prints a formatted log that includes the No., Category, Amount, and Description of each expense, similar to `viewMonthlyExpenses()`.
 - This log is useful for reviewing a detailed history of all expenses entered into the system.

- **Visual:**

```
Expense Tracking Menu
-----
1. Add Expenses
2. View Monthly Expenses
3. Calculate Total and Average Expenses
4. View Expense Log
5. Back to Main Menu
Enter your choice: 4

Expense Log:
+-----+-----+-----+-----+
| No. | Category | Amount | Description |
+-----+-----+-----+-----+
| 1 | labour | $56.00 | charge_of_temporary_labours |
| 2 | rod | $434.00 | fencing_rod |
| 3 | lunch | $6.00 | food_for_labours |
| 4 | fertilizer | $120.00 | crop_nutrients |
| 5 | seeds | $75.00 | various_seeds |
| 6 | pesticides | $90.00 | pest_control |
| 7 | water | $30.00 | irrigation |
| 8 | tools | $150.00 | farming_tools |
| 9 | transport | $200.00 | crop_transportation |
| 10 | maintenance | $45.00 | equipment_maintenance |
| 11 | Cement | $532.00 | This_is_paid_to_babari |
+-----+-----+-----+-----+
Press Enter to continue...|
```



4. Exit and Loading (Utility) Functions

Before running, the system loads the current data from the file by invoking the `loadData()` function and stores in the respective structures.

Before exiting, the system saves the current data to the file by invoking the `saveData()` function and frees dynamically allocated memory (crops and expenses). Frees the memory allocated for crops and expenses arrays using `free()` to prevent memory leaks.

4.1. `loadData()`:

This function loads previously saved crop and expense data from a file into memory before the program runs.

- **Details:**
 - Opens the file in read mode ("r"). If the file cannot be opened, an error message is displayed, and the function returns.
 - It reads the file line by line and determines whether it is in the Crops or Expenses section.
 - Depending on the section, it parses the data using `sscanf` to extract fields for crops or expenses. `sscanf()` function is use to get the different data types.
 - Dynamically allocates memory to store the crop or expense data, ensuring the arrays (crops and expenses) grow to accommodate new entries.
 - If memory allocation fails during this process, an error is displayed, and the file is closed.
 - After reading all data, the file is closed, and a success message is printed.
- **Visual:**

```
C:\Users\adhik>docker run -it swt-smart-farm-management
Data loaded successfully!
```

```
Welcome to Smart Farm Management System
-----
```

4.2. `saveData()`:

The program ensures all user data (crops, and expenses) is saved in a file named `farmerDetails.txt`. Upon exiting, the data is written to this file, allowing for future retrieval and updates.

- **Details:**
 - Opens the file in write mode ("w"). If the file cannot be opened, it prints an error message and returns.



- Writes the Crops section header, followed by each crop's details (name, area, yield, planting date, harvest date, and status).
- Writes the Expenses section header, followed by each expense's details (category, amount, and description).
- Closes the file after successfully writing the data and prints a success message.
- **Visual:**

```

Welcome to Smart Farm Management System
-----
1. Manage Crops
2. Irrigation Scheduling
3. Expense Tracking
4. Exit
Enter your choice: 4

Saving Data State Into File... Please Wait.
Data saved successfully!
Saved Successfully.

```

4.3 *lowerCase(char *s)*


This utility function converts all characters of a string to lowercase.

- **Details:**
 - Iterates through each character of the input string and converts it to lowercase using the `tolower()` function from the C standard library.
 - Returns the modified string.
- **Use Case:**
 - This function can be used to ensure uniformity in string comparisons, particularly when comparing or sorting strings in a case-insensitive manner.
 - This function can also be used to search different fields which we don't need to type exact.

Memory Management

- **Dynamic Arrays:** Arrays for fields, crops, and expenses grow dynamically using `realloc()` whenever new items are added. This ensures memory is used efficiently and only as much memory as needed is allocated.






```

1 // Dynamically allocating or reallocating memory for crops
2     crops = realloc(crops, sizeof(Crop) * (cropCount + 1));
3     if (crops == NULL)
4     {
5         perror("Failed to allocate memory for new crop");
6         holdingTerminal();
7         return;
8     }

```



```

1 //Dynamically allocating or reallocating memory for expenses
2     expenses = realloc(expenses, sizeof(Expense) * (expenseCount + 1));
3     if (expenses == NULL)
4     {
5         perror("Failed to allocate memory for new expense");
6         holdingTerminal();
7         return;
8     }

```

Error Handling

- **Incorrect Data-types:** User inputs are validated for correct types (e.g., ensuring numbers are entered for fields like amount in expenses, choosing menu numbers, and common things like percentage shouldn't be less than 0 or greater than 100 or amount can't be negative).
 - **File I/O Errors:** When reading from or writing to a file, errors are handled with `perror()` to provide meaningful error messages.
 - **Memory Allocation:** Functions check whether memory allocation (`malloc()` or `realloc()`) fails, and appropriate error messages are displayed using `perror()`.
-

Terminal Formatting

The program uses **ANSI escape codes (31-37)** to colorize output in the terminal, making the UI more user-friendly:

- Red (`\033[1;31m`) for error or important alerts.
-



- Green (\033[1;32m) for success messages.
- Yellow (\033[1;33m) for warnings or highlights.
- White/Cyan (\033[1;37m, \033[1;36m) for headers and table formatting.

These codes improve readability and provide visual feedback to the user.

Future Improvements

Here are some potential future improvements to this program:

- **Advanced Filtering and Sorting:** Allow users to filter or sort expenses and crops by specific attributes (e.g., category, water needs).
 - **Graphical User Interface (GUI):** Replace the command-line interface with a GUI for a more modern, user-friendly experience.
 - **Cloud Storage Integration:** Store data in the cloud (e.g., using a database) for better persistence and access across devices.
 - **Data Analysis:** Include additional analysis features such as predicting water needs based on weather forecasts or tracking expense trends over time.
 - **Improvement In Saving File:** Switch to a more structured file format like **JSON** for storing crops and expenses. This format is easier to parse and modify and is widely supported by many languages, making it easier to extend our system in the future.
-

General Queries:

Why sscanf() instead of fscanf() ?

- ✓ sscanf() to extract data from a string and also controlling buffer, flexibility in parsing. sscanf() parses the data from the buffer safely.

Why to use dynamic memory allocation like realloc(), calloc(), malloc() and free()?

- ✓ Dynamic allocation allows you to create and adjust memory usage on-the-fly during runtime, making the program more efficient and adaptable.

What's the advantage of error handling with perror()?

- ✓ perror() prints an error message associated with the last system call (like fopen()) that failed. This helps identify specific issues, such as failing to open a file due to permission errors or missing files.
-



References

1) Dynamic Memory Allocation:

- Use of `malloc()`, `realloc()`, and `free()`:
 - [*C Programming Tutorial*](#). (n.d.). *Dynamic Memory Allocation in C*.

2) File I/O and Array of C:

- *Computer Science Hand Book Class XII* by Bhanubhakta Regmi, Chandra Bilash Bhurtel.
- *Computer Science II Hand Book - (Grade XI)* by Buddha Publication

3) Basic C Programming:

- *Computer Science Hand Book Class XI* by Buddha Publication
- [*Computer Science II*](#) - (Grade XII) by Buddha Publication



