1. Abstract

The Recipe Management System is a console-based utility designed to assist users in organizing culinary data efficiently. Written in the C programming language, the project moves beyond static data storage by implementing dynamic memory allocation, allowing the system to scale according to user needs during runtime. The application adopts a persistent storage approach, utilizing file handling to serialize data into text files, ensuring that records are retained across sessions. Key computer science concepts demonstrated in this project include the manipulation of complex structures, pointer arithmetic, dynamic array resizing, and structured file I/O operations.

---

2. Problem Definition

2.1 Background

In domestic and small-scale commercial environments, recipe management is often performed manually using physical notes or unorganized text files. This leads to data redundancy, difficulty in modification, and the risk of physical data loss.

2.2 Problem Statement

The objective is to develop a lightweight "Command Line Interface" (CLI) application that solves the following issues:

1. Data Organization: Structured storage of recipe names, serving sizes, and detailed ingredient lists.
2. Scalability: The system must not be restricted by a hard-coded limit of recipes (e.g., fixed arrays) but should grow as the user adds data.
3. Persistence: Data must be saved to the hard drive to prevent loss upon program termination.
4. Resource Management: Efficient use of RAM by allocating memory only when required and freeing it upon deletion.

---

3. System Design

3.1 Data Structures

The system uses a relational structure design where a "Recipe" contains a pointer to a dynamic list of "Ingredients."

1. Structure: Ingredient This structure represents a single item required for a recipe.

- char* name: A pointer to a string for the ingredient name.
- double qty: Stores the amount needed.
- char unit[20]: Stores the unit of measurement (e.g., "grams", "cups").

2. Structure: Recipe This is the primary entity.

- int id: A unique identifier for indexing.
- char* name: A pointer to the recipe name.

- int servings: Metadata for the recipe.
- Ingredient* ing_list: A pointer acting as a dynamic array for ingredients.
- int ing_count: Tracks the current number of ingredients.
- int ing_alloc: Tracks the total memory allocated (capacity) to minimize reallocation overhead.

3.2 Functional Modules

The program is divided into modular functions to ensure maintainability:

- Input Helpers: get_str, get_int, get_double handle user input and prompt display.
- Memory Managers: check_db_size and add_ingredient handle realloc logic.
- Logic Controllers: create_recipe, delete_recipe, find_recipe.
- File Handlers: save_data and load_data.

---

4. Implementation Details

4.1 Dynamic Memory Allocation Strategy

To ensure memory efficiency, the program uses a geometric expansion strategy.

- Global Recipe Array: When the all_recipes array is full, the check_db_size() function is triggered. It doubles the current capacity (e.g., 10 -> 20 -> 40) using realloc().
- Ingredient Lists: Similarly, every recipe structure maintains its own capacity for ingredients. This ensures that a recipe with 50 ingredients does not waste space in a recipe that only needs 3.

Snippet: Capacity Expansion

C

```c
if (recipe_count >= recipe_cap) {
    int new_cap = (recipe_cap == 0) ? 10 : recipe_cap * 2;
    all_recipes = realloc(all_recipes, sizeof(Recipe) * new_cap);
    // Error checking is performed here to ensure memory was granted
}
```

4.2 Data Persistence and Parsing

The system uses a custom text-based database format. Two files are used to simulate a relational database:

1. recipes.dat: Stores the header information.
2. ingredients.dat: Stores the detailed items, linked to the main file via the Recipe ID.

Delimiter Logic: The pipe character (|) is used as a delimiter instead of spaces. This allows the program to handle multi-word inputs (e.g., "Ice Cream") correctly using fscanf pattern matching.

Snippet: Loading Logic

C

```c
// Reads until a pipe is found, handling spaces in names
fscanf(fp, "%d|%99[^|]|%d\n", &id, name, &serv);
```

4.3 The Deletion Algorithm

Deleting an entry from a dynamic array requires shifting elements to close the "gap" created by the removed item.

1. Search: The program iterates to find the index of the target ID.
2. Memory Cleanup: free() is called on the name string and the inner ingredient array to prevent memory leaks.
3. Shift: A loop moves all_recipes[i+1] into all_recipes[i] from the deletion point to the end of the list.
4. Resize: The count is decremented.

---

5. Testing and Results

The application underwent "Black Box Testing" to validate functionality against requirements.

5.1 Test Case: Input Handling

- Objective: Verify that strings with spaces are accepted.
- Input: Recipe Name: "Chicken Curry", Ingredient: "Curry Powder".
- Result: The system correctly stored the full strings without truncating at the whitespace.

5.2 Test Case: ID Generation

- Objective: Verify unique IDs are generated even after deletions.
- Procedure: Created 3 recipes (IDs 1, 2, 3). Deleted ID 2. Created a new recipe.
- Result: The get_next_id() function scanned the existing IDs (1, 3) and correctly assigned ID 4, avoiding conflicts.

5.3 Test Case: File I/O

- Objective: Verify data integrity.
- Procedure: Added complex fractional quantities (e.g., 2.5 Cups). Saved and exited.
- Result: Upon reloading, the double data type correctly preserved the precision (2.50).

---

6. Conclusion and Future Scope

6.1 Conclusion

The project successfully implements a fully functional Recipe Management System. It fulfills the primary requirements of adding, viewing, deleting, and persisting data. The use of struct and realloc demonstrates an understanding of advanced C memory management, while the fscanf implementation shows capability in parsing structured text files.

6.2 Limitations

- Sequential Search: The search function uses linear time complexity O(n), which is efficient for small datasets but may slow down with thousands of recipes.
- Console Only: The lack of a GUI makes it less accessible to non-technical users.

6.3 Future Work

1. Search Functionality: Implement a search-by-ingredient feature to suggest recipes based on available stock.
2. Sorting: Implement QuickSort or MergeSort to allow listing recipes alphabetically or by ID.
3. Error Handling: Improve input validation to handle cases where a user enters text instead of a number for quantity.

---

7. References

1. Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language (2nd ed.). Prentice Hall.
2. Prata, S. (2013). C Primer Plus (6th ed.). Addison-Wesley Professional.
3. ISO/IEC. (1999). ISO/IEC 9899:1999 - Programming languages — C.