ELEC 278 Final Assignment Report

Mark Wang

12/06/2023

I, Mark Wang, attest that all of the materials which I am submitting for this assignment are my own and were written solely by me. I have cited in this report any sources, other than the class materials, which I used in creating any part of this assignment. Furthermore, this work adheres to the policy on generative artificial intelligence as documented in the instructions.

Executive Summary

The provided C code implements a basic spreadsheet model with functions for initializing, updating, and interacting with cells. The model supports three types of cell values: text, numbers, and formulas. Key functions include model_init() for initializing the spreadsheet, set_cell_value() for updating cell values based on user input, update_dependents() for recalculating dependent cells when formulas change, and clear_cell() for resetting cell values. The evaluate_formula() function parses and calculates formula expressions. Overall, the code establishes a simple spreadsheet structure with dynamic cell dependencies, providing foundational functionality for a basic spreadsheet application. The main data structures used from ELC 278 are 2D arrays and linked lists.

Design Proposal

The assignment given was to create a spreadsheet, where each cell within the spreadsheet could store either an input string, a double datatype which means any number, as well as a formula which would be two cell references added together, with the input beginning with the operator "=". Intuitively, a spreadsheet appears to be very similar to a 2D array, where the spreadsheet needed to a 10 rows by 7 columns grid of cells, resulting in a total number of 70 columns. As this array is a set number, and the number of cells do not change, it was determined that initializing a two dimensional array to represent the spreadsheet would be the best course of action. Within each cell, a custom structure was declared capable of holding a string (declared as a char pointer), the cell's type of value (double datatype, string, or formula), the actual formula inputted, the value that the formula calculates (which will be necessary for displaying the formula correctly to users on the interface), and a linked list node for the scenario where a linked list will be required to update dependent cells. This design allows flexibility in accommodating various data types within the spreadsheet cells. They also help fulfill the functional requirements which will be discussed below. The original brainstorming on paper can be seen below in Figure 1. A screenshot of custom structures implemented into the code can be seen below in Figure 2.

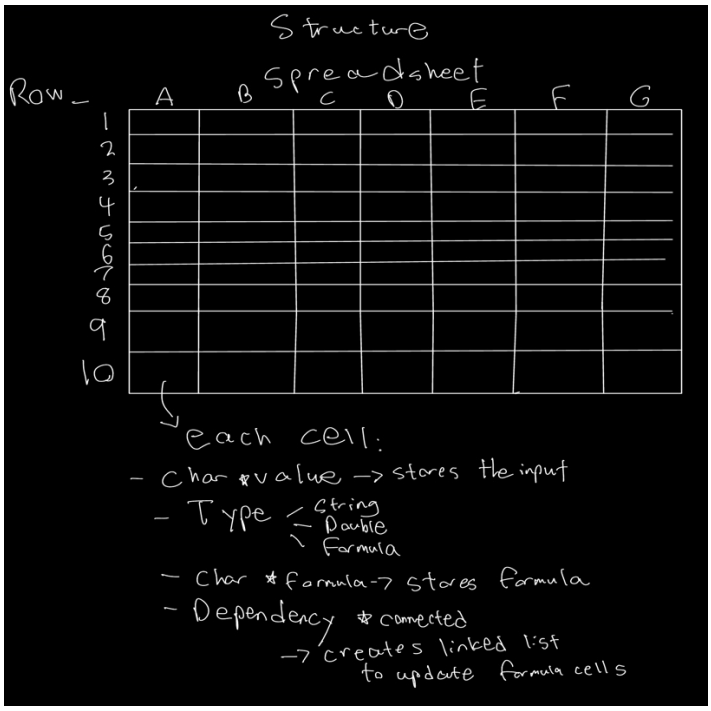


*Figure 1: Original brainstorming for selecting data structures. It should be noted that another char *result was added in later on in development due to the need for printing out the formula when the cell is selected by the user*

```c
// Enumeration to represent the different types of cell values
typedef enum {
    TEXT,
    NUMBER,
    FORMULA
} ValueType;

// Structure to represent a node in the dependency linked list
typedef struct DependencyNode {
    ROW row;
    COL col;
    struct DependencyNode* next;
} DependencyNode;

// Structure to represent a cell in the spread
typedef struct {
    char *cell_value;
    char *result;
    ValueType type;
    char *formula;
    DependencyNode *dependencies;
} spreadCell;

// Double pointer to represent the spread
spreadCell **spread;

// Function to check if a given text is a number
```

Figure 2: Global variables/defining the necessary structs and variables for development of the selected data structures

The functional requirements and how they were fulfilled with the proposed design solution are listed below:

1. Be able to navigate between cells and clear cells when desired.
    a. The 2D array allocates the appropriate memory and the indices to navigate and clear any cells that the user desires. This will be implemented with the starter code given to allow users to navigate with their keyboard, as well as a clear cell function which will be discussed later in implementation.
2. User input is interpreted appropriately as a number, string, or formula (two cell references added up and input begins with the operator "=")
    a. The custom structure declared for each cell results in the ability to recognize and store each cell as a certain type, which can be a number, string, or formula. In the code shown above, there is only one datatype declared for strings and numbers. This is because any input text that is not a formula can be first stored as a string and manipulated later on with other functions discussed below in implementation.
3. A textual representation of the cell will be displayed in the top left of the interface, which is editable by the user and user can edit the value. When it is a formula, it should display as the formula and not the calculated value on the top left.
    a. Declaring a custom char *result will help later on with keeping a copy of the formula, so that the cell can store both the calculated value for the formula but also the display of the formula on the top left of the user interface
4. When the value of a cell changes, the displayed contents of any dependent cells update as well, which are cells that have formulas that rely on the cell whose value changed.
    a. Using a linked list, which was declared with the DependencyNode seen above will help update any dependent cells

Expanding further on the linked list for updating dependencies, DependencyNode can also be seen declared above in Figure 1, which helps initialize a linked list to establish dependencies between cells, specifically for formula that connect multiple cells. A linked list was chosen as it facilitates the dynamic management of relationships between cells, ensuring that changes in one cell can propagate and update dependent cells. Furthermore, this design is aligned with Non-Functional Requirement 1 (NFR 1), emphasizing efficiency and scalability. The 2D array enables constant-time access to cells, and the use of

linked lists ensures a dynamic and scalable solution for handling dependencies, supporting the overall performance and extensibility of the spreadsheet application.

Implementation

model_init()

The model_init() function was used to initialize the 2D array that will be used to store all datatypes, including strings, numbers, and any formulas. It can be seen that the function allocates memory for the rows and columns separately to initialize the appropriate memory. The nested for loop allows for iteration through all rows and columns, and each cell in the grid is initialized as NULL. The default type for all cells is also text so users can input anything they'd like to and formulas will be searched for through looking at the first character of the input. Dependencies are also set as NULL to ensure that there are no dependencies unless required. The time complexity of the algorithm would be O(Rows x Columns) which would be O(70) as there are 10 rows and 7 columns in total.

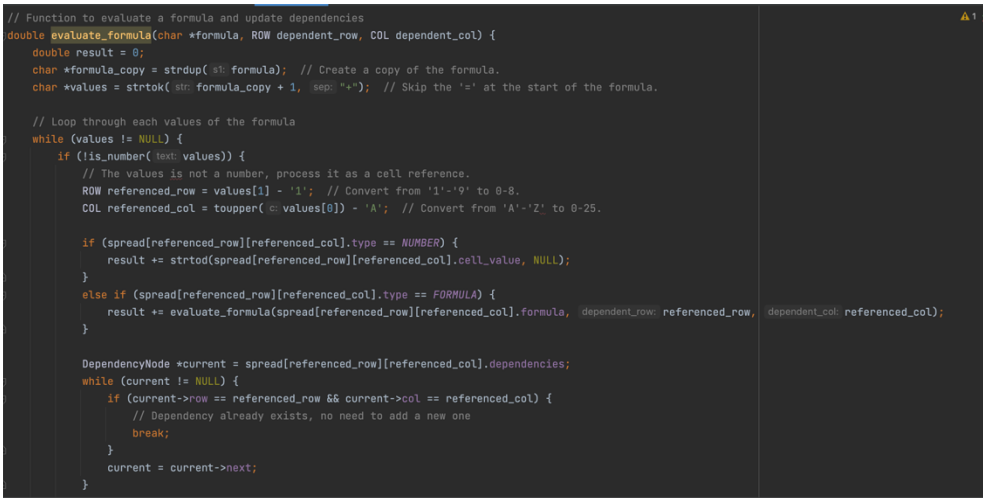
```
// Function to initialize the model and the spread
void model_init() {
    spread = malloc( size: ROWS * sizeof(spreadCell *));
    for (int i = 0; i < ROWS; i++) {
        spread[i] = malloc( size: COLS * sizeof(spreadCell));
        for (int j = 0; j < COLS; j++) {
            spread[i][j].cell_value = NULL;
            spread[i][j].type = TEXT;
            spread[i][j].dependencies = NULL;
        }
    }
}
```

Figure 3: model_init function

evaluate_formula()

The evaluate_formula function plays a large role in computing the result of a cell's formula. Initially, it creates a copy of the formula using the function strdup*, skipping the '=' character at the beginning with strtok**. The algorithm then iterates through each component of the formula, distinguishing between numbers and cell references. For cell references, it converts the alphanumeric representation to corresponding row and column indices. The function accumulates the result by recursively evaluating dependent cells, considering both direct numerical values and other formula cells. To manage

dependencies, it updates the linked list for each referenced cell. The function ensures memory cleanup by freeing the copied formula string. The time complexity is linear (O(N)), where N is the length of the formula, reflecting the sequential parsing process. The recursive evaluations maintain efficiency even for intricate formulas within the spreadsheet.



```c
// Function to evaluate a formula and update dependencies
double evaluate_formula(char *formula, ROW dependent_row, COL dependent_col) {
    double result = 0;
    char *formula_copy = strdup( s1: formula);  // Create a copy of the formula.
    char *values = strtok( str: formula_copy + 1,  sep: "+");  // Skip the '=' at the start of the formula.

    // Loop through each values of the formula
    while (values != NULL) {
        if (!is_number( text: values)) {
            // The values is not a number, process it as a cell reference.
            ROW referenced_row = values[1] - '1';  // Convert from '1'-'9' to 0-8.
            COL referenced_col = toupper( c: values[0]) - 'A';  // Convert from 'A'-'Z' to 0-25.

            if (spread[referenced_row][referenced_col].type == NUMBER) {
                result += strtod(spread[referenced_row][referenced_col].cell_value, NULL);
            }
            else if (spread[referenced_row][referenced_col].type == FORMULA) {
                result += evaluate_formula(spread[referenced_row][referenced_col].formula,  dependent_row: referenced_row,  dependent_col: referenced_col);
            }

            DependencyNode *current = spread[referenced_row][referenced_col].dependencies;
            while (current != NULL) {
                if (current->row == referenced_row && current->col == referenced_col) {
                    // Dependency already exists, no need to add a new one
                    break;
                }
                current = current->next;
            }
```

Figure 4: Part 1 of evaluate formula function

```c
                    // Dependency already exists, no need to add a new one
                    break;
                }
                current = current->next;
            }

            // If the dependency doesn't exist, add a new one
            if (current == NULL) {
                DependencyNode *newNode = malloc( size: sizeof(DependencyNode));
                newNode->row = dependent_row;
                newNode->col = dependent_col;
                newNode->next = spread[referenced_row][referenced_col].dependencies;
                spread[referenced_row][referenced_col].dependencies = newNode;
            }
        }
        else {
            // The values is a number, add it to the result
            result += strtod(values, NULL);
        }
        values = strtok( str: NULL,  sep: "+");
    }
    free(formula_copy);  // Free the copied string
    return result;
}
```

Figure 5: Second part of evaluate formula function

*strdup reference: https://www.geeksforgeeks.org/strdup-strdndup-functions-c/

*strtok reference: https://cplusplus.com/reference/cstring/strtok/

update_dependents()

As a functional requirement is that cells will update accordingly to any dependencies, the update_dependents function iterates through the linked list of dependencies for a given cell at the specified row and column. For each dependent cell, if its type is a formula, it recalculates its computed value using the evaluate_formula function, which will be discussed below. The result is then stored, and the display is updated accordingly using update_cell_display, a built in function given in starter code. This process ensures that any changes in the cell at the specified row and column trigger an update cascade, propagating through all dependent cells. The function manages memory appropriately by freeing the previously stored result before updating it with the newly computed value, thereby maintaining the integrity of the spreadsheet model. The time complexity of the algorithm would be O(n), where n is the number of dependencies.

```
// Function to update dependents of a cell
void update_dependents(ROW row, COL col) {
    DependencyNode *current = spread[row][col].dependencies;
    while (current != NULL) {
        if (spread[current->row][current->col].type == FORMULA) {
            // The dependent cell is a formula cell, update its computed value.
            double result = evaluate_formula(spread[current->row][current->col].formula, dependent_row: current->row, dependent_col: current->col);
            if (spread[current->row][current->col].result != NULL) {
                free(spread[current->row][current->col].result);
            }
            char *result_str = malloc( size: 20 * sizeof(char));  // Allocate enough space for a double.
            sprintf(result_str, "%.1f", result);
            spread[current->row][current->col].result = result_str;
            update_cell_display(current->row, current->col, text: spread[current->row][current->col].result);
        }
        current = current->next;
    }
}
```

Figure 6: update_dependents() function

set_cell_value()

The set_cell_value function is the primary function that handles all the input and updates for a cell at the specified row and column. It first frees any existing values in the cell to prevent memory leaks. Depending on the input text, it determines whether it's a formula, number, or text. Formulas will be

determined if the first character is a “=” operator. For formulas, it calculates the result using the evaluate_formula function, discussed above, updates the cell's values, and triggers the necessary display and dependency updates. It can be seen there is both a cell_value and result, there are two different components here to ensure that in the user interface, when the user navigates to a cell with a formula in it, the top left area will still show the original formula inputted and not just the calculated value. If the input is a number or text, it directly updates the cell's values. The function manages memory efficiently by freeing existing values, ensuring proper storage of new data, and avoiding memory leaks or overwrites. The time complexity of the algorithm would be O(n).

```c
// Function to set the value of a cell
void set_cell_value(ROW row, COL col, char *text) {
    // Free existing values in the cell
    if (spread[row][col].cell_value != NULL) {
        free(spread[row][col].cell_value);
    }
    if (spread[row][col].result != NULL) {
        free(spread[row][col].result);
    }

    // Check if the input text is a formula or a number
    if (text[0] == '=') {
        // The input text is a formula
        spread[row][col].type = FORMULA;
        spread[row][col].formula = strdup( s1: text);
        double result = evaluate_formula( formula: text,  dependent_row: row,  dependent_col: col);
        char *result_str = malloc( size: 20 * sizeof(char));  // Allocate enough space for a double.
        sprintf(result_str, "%.1f", result);
        spread[row][col].cell_value = strdup( s1: text);
        spread[row][col].result = result_str;
    } else {
        // The input text is a number or text
        if (is_number(text)) {
            spread[row][col].type = NUMBER;
            char *value_str = malloc( size: 20 * sizeof(char));  // Allocate enough space for a double.
            sprintf(value_str, "%.1f", strtod(text, NULL));
            spread[row][col].cell_value = value_str;
            spread[row][col].result = strdup( s1: value_str);
```

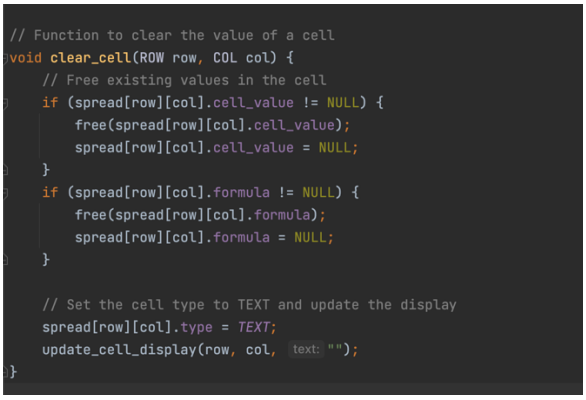
Figure 7: First part of set_cell_value() function

```c
        } else {
            spread[row][col].type = TEXT;
            spread[row][col].cell_value = strdup( s1: text);
            spread[row][col].result = strdup( s1: text);
        }
        spread[row][col].formula = NULL;
    }

    // Update the display and dependents
    update_cell_display(row, col,  text: spread[row][col].result);
    update_dependents(row, col);

    // Free the input text
    free(text);
}
```

Figure 8: Second part of set_cell_value() function

clear_cell ()

The clear_cell() function is responsible for resetting the specified cell at the given row and column. It begins by freeing any existing values in the cell, such as cell values and formulas, to prevent memory leaks. The function sets the cell type back to the default TEXT, indicating an empty cell, and updates the display accordingly. This process ensures a clean slate for the cell, making it ready for new input. The function is essential for user interactions when users want to clear a cell's content or reset it to a default state, contributing to the overall usability and maintenance of the spreadsheet model. The time complexity of the algorithm is O(1).

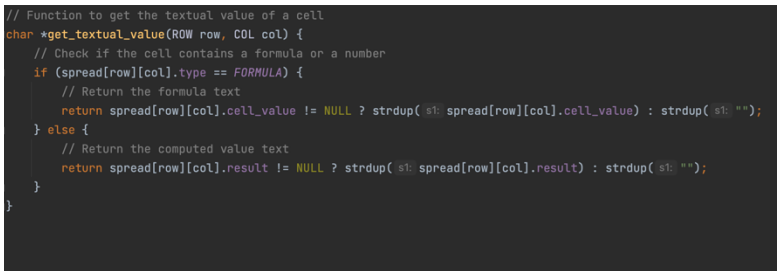


```
// Function to clear the value of a cell
void clear_cell(ROW row, COL col) {
    // Free existing values in the cell
    if (spread[row][col].cell_value != NULL) {
        free(spread[row][col].cell_value);
        spread[row][col].cell_value = NULL;
    }
    if (spread[row][col].formula != NULL) {
        free(spread[row][col].formula);
        spread[row][col].formula = NULL;
    }

    // Set the cell type to TEXT and update the display
    spread[row][col].type = TEXT;
    update_cell_display(row, col,  text: "");
}
```

Figure 9: clear_cell() function

get_textual_value ()

The get_textual_value function provides the textual representation of a cell's content, considering whether it contains a formula or a computed value. Depending on the cell type, it returns either the formula or the computed value text, as this is the functional requirement 3. This function operates in constant time, with a time complexity of O(1). Its efficiency remains consistent regardless of the complexity of the formula or the length of the computed value. This efficiency makes it a reliable choice for fetching textual representations in a spreadsheet, contributing to a seamless user experience.



```
// Function to get the textual value of a cell
char *get_textual_value(ROW row, COL col) {
    // Check if the cell contains a formula or a number
    if (spread[row][col].type == FORMULA) {
        // Return the formula text
        return spread[row][col].cell_value != NULL ? strdup( s1: spread[row][col].cell_value) : strdup( s1: "");
    } else {
        // Return the computed value text
        return spread[row][col].result != NULL ? strdup( s1: spread[row][col].result) : strdup( s1: "");
    }
}
```

Figure 10: get_textual_value() function

Testing

Testing was done through a plan that could test if the code met all the functional requirements. Below is a sequential list of how testing was completed.

1. The first test was to navigate between all the cells with the keyboard and access any cell. This was quite simple as most of the navigation was built in the starter code, so no screenshot is necessary.
2. To test if cells could store all strings or numbers, random numbers and letters were placed all around the spreadsheet to determine if it could store all datatypes appropriately. Figure below shows how the test was successful in doing so.

B7	A	B	C	D	E	F	G
1	1.0		3.0				awdawd
2					8.0		
3		2.0					0.0
4	3.5		2.0				
5					123123.0		
6	awd	test					
7			ad\				jrsthx
8	test					adw	
9							
10			10.0				

Press Ctrl+C to exit.

Figure 11: Test Number 2

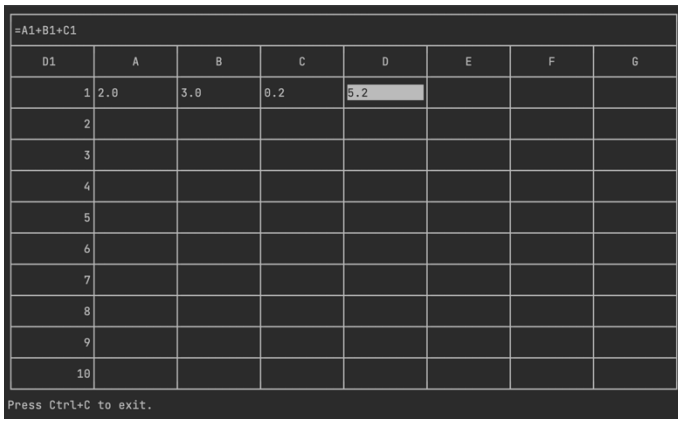
3. To test formulas, a simple formula was set up where A1 and A2 have values of 1 and 2 respectively, and A3 was set to “=A1+A2”, which printed out 3 while maintaining the formula on the top left of the interface.

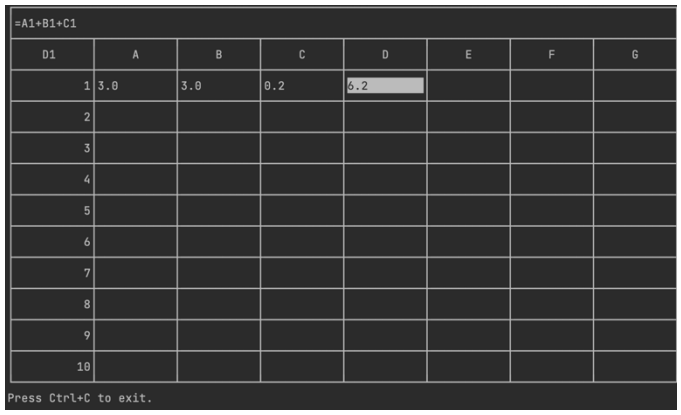
=A1+A2

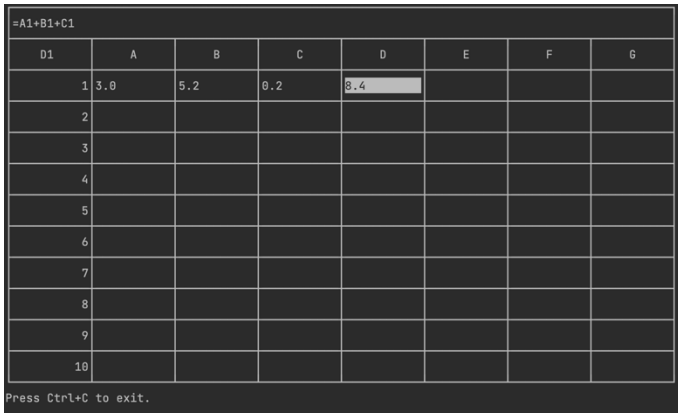
A3	A	B	C	D	E	F	G
1	1.0						
2	2.0						
3	3.0						
4							
5							
6							
7							
8							
9							
10							

Press Ctrl+C to exit.

4. To test dependencies and more complex formulas, A1 was set to 2, B1 was set to 3, C1 was set to 0.2, and D1 was set to “=A1+B1+C1”. Figure below shows the result of this. After, A1 was now changed to 3, which Figure below shows D1 changing appropriately. After this B1 was set to 5.2, and D1 changed appropriately as well. This indicates all dependencies are functional.







5. Finally, to test dependencies on a deeper level, A1 was set to 3.5, A2 was set to “=A1” and A3 was set to “=A2”. A3 depends on A2 which depends on A1. When A1 was changed, A2 and A3

changed as expected seen below in Figure