TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies Department
of Computer Systems

Ihor Kornitskyi 230628MVEB

Matrix Formation and Display Using Subroutines in C

Homework II

Supervisor: Vladimir Viies

Tallinn 2024

Author’s declaration of originality

I confirm that I am the sole author of this thesis. All sources and literature references have been properly cited, and the work has not been submitted for examination elsewhere.

Author: Ihor Kornitskyi

22.09.2024

Table of contents

List of figures

1 Task description

The task was to create a C program that reads a specific number of elements into an array A, organizes these elements into a matrix B based on a user-defined row length k, and then displays the matrix row by row. If the total elements in array A didn't perfectly fit the matrix, the remaining slots in the matrix needed to be filled with zeros. Additionally, the solution had to be modular, utilizing multiple subroutines (functions) to handle different parts of the process.

2 Solution description

I implemented the program by breaking it into five subroutines (functions) to handle each part of the task efficiently. This approach made the code more organized, easier to manage, and ensured that the program met the assignment requirements for using subroutines.

1. **User Input:**

The program first prompts the user for three main inputs:

- An integer n representing the number of elements in array A. The user is asked to enter a value between 2 and 9.
- n real numbers (floating-point) to fill array A.
- A positive integer k, representing the number of columns for matrix B.

The program checks for valid inputs at each step and displays an error message if the inputs are out of range or invalid.

2. Data Storage:

- The program stores n elements entered by the user in a one-dimensional array A.
- It then creates a two-dimensional matrix B based on the user-defined value k. The matrix B has rows calculated as **(n+k−1) /k (n + k - 1) / k (n+k−1)/k**, ensuring enough space to store all elements from A, with extra positions filled with zeros if necessary.

3. Results:

The final output is the matrix B, displayed row by row on the screen. Each row shows the elements from array A in their respective positions, and any remaining positions in the last row are filled with zeros to maintain the matrix's shape.

2.1 Program flow

Here's how it goes step-by-step:

1. **Input the Number of Elements (`readNumberOfElements`)**:

    - First, I used the `readNumberOfElements` function to ask the user for the number of elements, n, that would be stored in array A. The function checked that n was between 2 and 9. If it wasn't, the program displayed an error message and stopped running.

2. **Reading the Array Elements (`readArrayElements`)**:

    - Once I had a valid n, the program called the `readArrayElements` function. This function prompted the user to input nnn floating-point numbers, which were stored in array A. This step ensured that the data for matrix formation was ready.

3. **Input the Row Length kkk (`readRowLength`)**:

    - After getting the elements of A, the program called `readRowLength`. This function asked the user to enter a positive integer k for the matrix's row length. It validated the input to make sure k was greater than zero; otherwise, an error message was displayed, and the program stopped.

4. **Creating the Matrix (`formMatrix`)**:

    - With n, k, and the elements of A ready, the `formMatrix` function was used to fill a matrix B with the elements from A. The function iterated over the elements of A and placed them into matrix B row by row. If there were any leftover positions (meaning n wasn't a perfect multiple of k), the function filled those spots with zeros.

5. **Displaying the Matrix (`displayMatrix`)**:

    - Finally, the program called the `displayMatrix` function to output matrix B row by row on the screen. This function displayed the matrix with two decimal places for each element, making the output neat and easy to read.

Why This Flow Works: By separating each part of the program into its own function, the flow became much clearer and easier to manage. It allowed me to focus on one task at a time, ensuring that each part worked correctly before moving on to the next. This approach also made the program easy to debug and modify if needed.

2.2 Algorithm

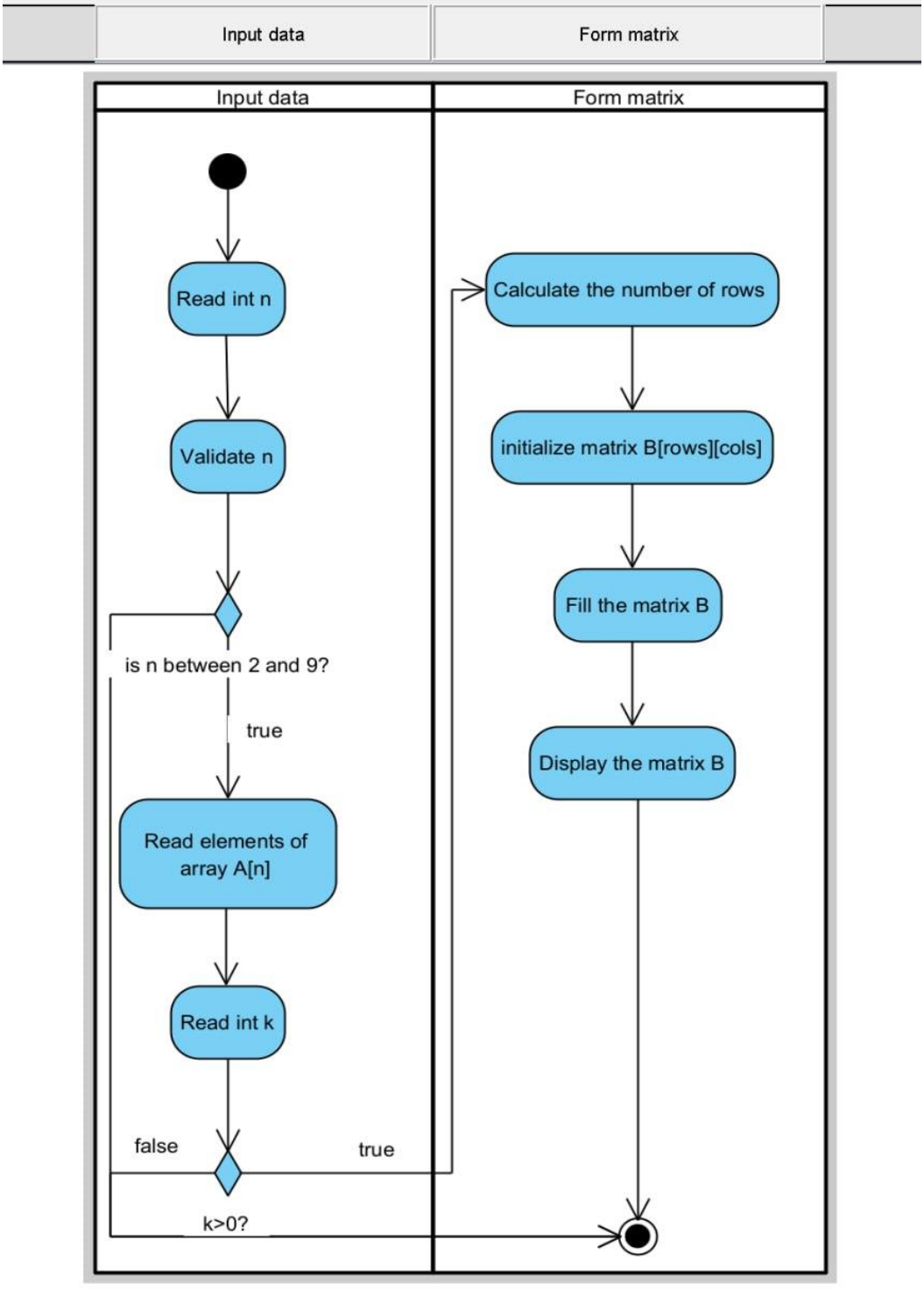

Figure 2. UML diagram of the application algorithm from Visual Paradigm.

3 Summary

In this homework, I created a modular C program that reads user inputs, stores the data in an array, and then formats that data into a matrix. The solution met all requirements by using subroutines for each step, making the program easier to read, understand, and maintain. I learned how to efficiently manage data using arrays and matrices and how to structure my program using functions for a cleaner, more organized approach. This experience improved my understanding of working with 2D arrays and user input handling in C.

Any feedback is welcome and will be applied to improve the code!

References

[1] Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language* (2nd ed.). Prentice Hall.

Appendix 1 – Screenshots (Code & Output)

```c
#include <stdio.h>

// Function prototypes
int readNumberOfElements();
void readArrayElements(double[], int);
int readRowLength();
void formMatrix(double[], double[][10], int, int, int);
void displayMatrix(double[][10], int, int);

// Main function
int main() {
    // Step 1: Read the integer n
    int n = readNumberOfElements();
    if (n == -1) return 1; // Exit if invalid input

    // Step 2: Read the elements of array A
    double A[n];
    readArrayElements(A, n);

    // Step 3: Read the positive integer k
    int k = readRowLength();
    if (k == -1) return 1; // Exit if invalid input

    // Step 4: Calculate the number of rows needed
    int rows = (n + k - 1) / k;
    double B[rows][10]; // Assuming a maximum column size of 10

    // Form the matrix B using the function
    formMatrix(A, B, n, k, rows);

    // Step 5: Display the matrix B
    displayMatrix(B, rows, k);

    return 0;
}

// Function definitions

// Function to read the number of elements
int readNumberOfElements() {
    int n;
    printf("Enter the number of elements in array A (2 <= n < 10): ");
    scanf("%d", &n);
    if (n < 2 || n >= 10) {
        printf("Invalid input. n must be between 2 and 9.\n");
        return -1; // Return -1 to indicate an error
    }
    return n;
}

// Function to read array elements
void readArrayElements(double A[], int n) {
    printf("Enter %d elements for array A:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%lf", &A[i]);
    }
}

// Function to read the row length k
int readRowLength() {
```

```c
    int k;
    printf("Enter a positive integer k: ");
    scanf("%d", &k);
    if (k <= 0) {
        printf("Invalid input. k must be a positive integer.\n");
        return -1; // Return -1 to indicate an error
    }
    return k;
}

// Function to form the matrix B
void formMatrix(double A[], double B[][10], int n, int k, int rows) {
    int index = 0;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < k; j++) {
            if (index < n) {
                B[i][j] = A[index++];
            } else {
                B[i][j] = 0.0; // Fill remaining cells with 0
            }
        }
    }
}

// Function to display the matrix B
void displayMatrix(double B[][10], int rows, int k) {
    printf("Matrix B:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < k; j++) {
            printf("%.2lf ", B[i][j]);
        }
        printf("\n");
    }
}
```

Output

```
/tmp/vcNtLn4kJ6.o
Enter the number of elements in array A (2 <= n < 10): 7
Enter 7 elements for array A:
1.5 2.3 3.1 4.0 5.6 6.7 7.8
Enter a positive integer k: 3
Matrix B:
1.50 2.30 3.10
4.00 5.60 6.70
7.80 0.00 0.00
```