Study Time: 5.5 hours

Exercise Time: 2.5 hours

Documentation of Day 23

Exercise 1-20.

Write a program detab that replaces tabs in the input with the proper number of blanks to space to the next tab stop. Assume a fixed set of tab stops, say every n columns. Should n be a variable or a symbolic parameter?

Source Code:

```c
#include <stdio.h>

#define TAB_WIDTH 4  // Set the tab stop value as a symbolic parameter

int main(void) {
    int c;
    int position = 0;

    while ((c = getchar()) != EOF) {
        if (c == '\t') {
            int spaces = TAB_WIDTH - (position % TAB_WIDTH);
            int i;
            for (i = 0; i < spaces; i++) {
                putchar('*'); /*Determines that the tabs are replaced by tabs*/
                position++;
            }
        } else if (c == '\n') {
            putchar(c);
            position = 0;
        } else {
            putchar(c);
            position++;
        }
    }

    return 0;
}
```

Key Points of my approach to this solution:

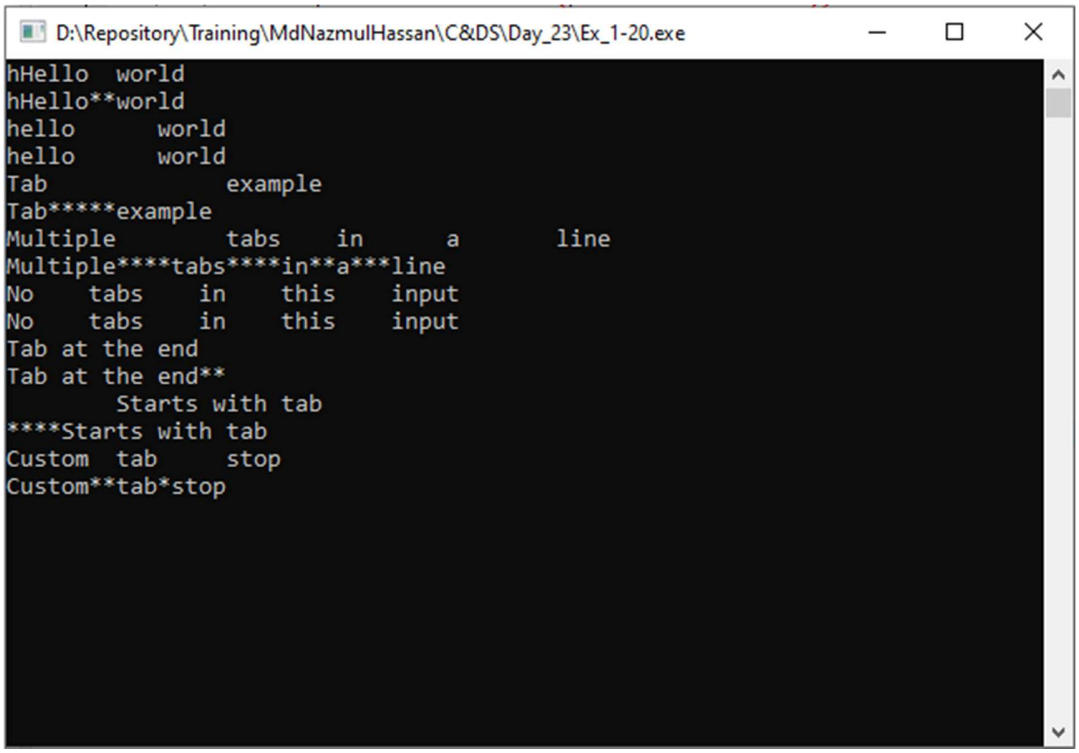
The **detab** program replaces tab characters with spaces based on a symbolic parameter, **TAB_WIDTH**. Here are the key features:

1. **Symbolic Parameter: TAB_WIDTH** represents the tab stop value, which can be easily modified to any desired number of columns.
2. **Input Processing:** The program **reads input** character by character using **getchar().**
3. **Handling Tabs:** When **a tab character is encountered**, the **program calculates the number of spaces needed** to **reach the next tab stop using TAB_WIDTH** and **outputs the required spaces**.
4. **Handling Newlines: Newline characters** are **outputted as is**, and the **position is reset to 0.**
5. **Other Characters: Non-tab characters** are **outputted normally**, and the position is incremented.
6. **Fixed Tab Stops:** By using **TAB_WIDTH**, the program assumes a fixed set of tab stops at every **TAB_WIDTH columns**.

Important Note:

- In this exercise, the **requirement specifies a fixed set of tab stops**, meaning that the **tab stops are predetermined at regular intervals**. Therefore, in this case, **n should be a symbolic parameter rather than a variable.**
- **In my program, TAB_STOP** represents a **tab stop occurring every 4 columns**. **By using this symbolic parameter throughout the code, we can easily modify the value if needed**, without **having to search for and change multiple occurrences of the value manually**. **This could be an enhanced version of this detab program.**

Output:



Exercise 1-22.

Write a program to ``fold'' long input lines into two or more shorter lines after the last non-blank character that occurs before the n-th column of input. Make sure your program does something intelligent with very long lines, and if there are no blanks or tabs before the specified column.

Source Code:

```c
#include <stdio.h>
#include <string.h>

#define MAXLINE 10000

#define TRUE (1 == 1)
#define FALSE !TRUE

int get_line(char line[], int max_line_len);
void fold_line(char line[], char fold_str[], int n_break);
int find_longest_word(char line[]);

int main(void)
{
    char line[MAXLINE];
    char fold_str[MAXLINE];

    while (get_line(line, MAXLINE) > 0)
    {
        int breaking_point = find_longest_word(line);
        fold_line(line, fold_str, breaking_point);
        printf("%s", fold_str);
    }

    return 0;
}
```

```c

int get_line(char line[], int max_line_len)
{
    int c, i = 0;

    while (i < max_line_len - 1 && (c = getchar()) != EOF && c != '\n')
    {
        line[i++] = c;
    }

    if (c == '\n')
    {
        line[i++] = c;
    }

    line[i] = '\0';

    return i;
}
```

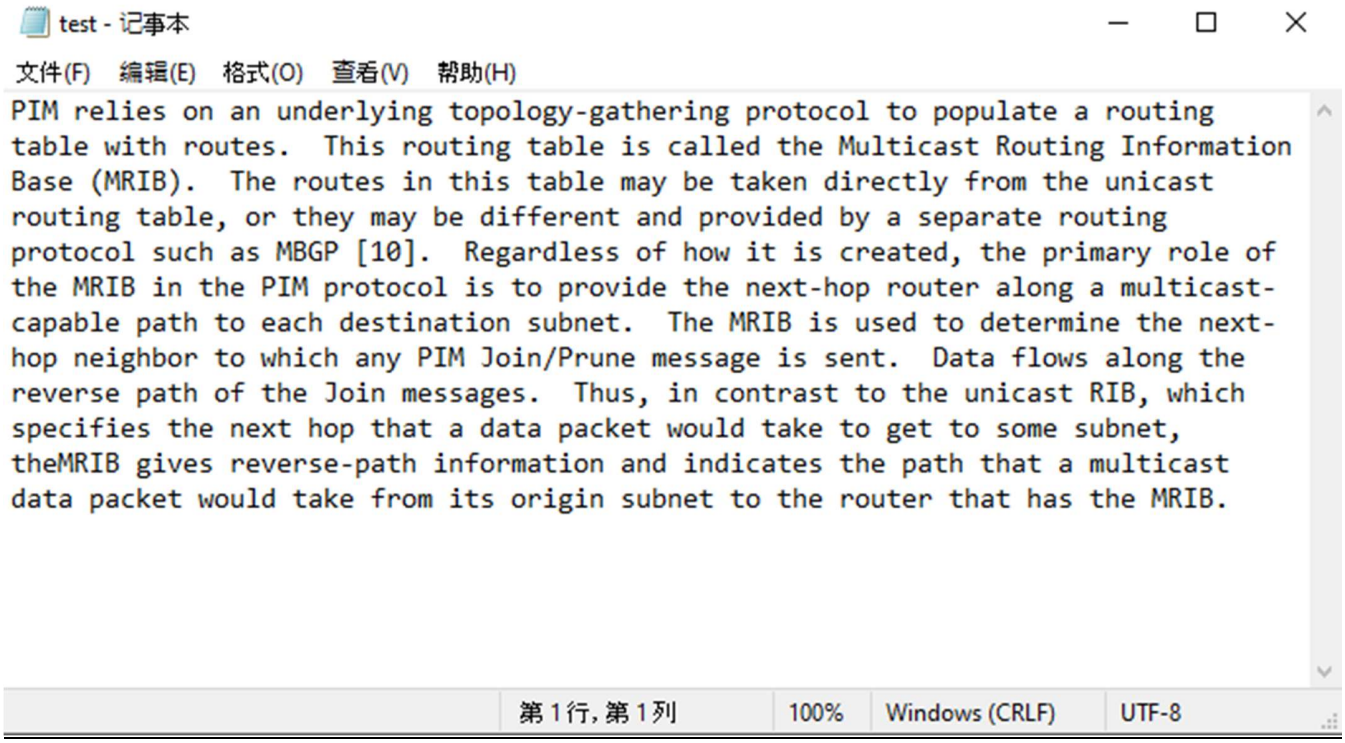
```c
void fold_line(char line[], char fold_str[], int n_break)
{
    int i, j;
    int column = 0;
    int last_blank = -1;

    for (i = 0, j = 0; line[i] != '\0'; ++i, ++j)
    {
        fold_str[j] = line[i];

        if (fold_str[j] == '\n')
        {
            column = 0;
            last_blank = -1;
        }

        column++;

        if (fold_str[j] == ' ' || fold_str[j] == '\t')
            last_blank = j;

        if (column == n_break)
        {
            if (last_blank != -1)
            {
                fold_str[last_blank] = '\n';
                column = j - last_blank;
                last_blank = -1;
            }
            else
            {
                fold_str[j++] = '-';
                fold_str[j] = '\n';

                column = 0;
            }
        }
    }

    fold_str[j] = '\0';
}
```

```c
int find_longest_word(char line[])
{
    int longest_word_length = 0;
    int current_word_length = 0;
    int i;

    for (i = 0; line[i] != '\0'; i++)
    {
        if (line[i] == ' ' || line[i] == '\t' || line[i] == '\n')
        {
            if (current_word_length > longest_word_length)
            {
                longest_word_length = current_word_length;
            }
            current_word_length = 0;
        }
        else
        {
            current_word_length++;
        }
    }

    if (current_word_length > longest_word_length)
    {
        longest_word_length = current_word_length;
    }

    return longest_word_length;
}
```

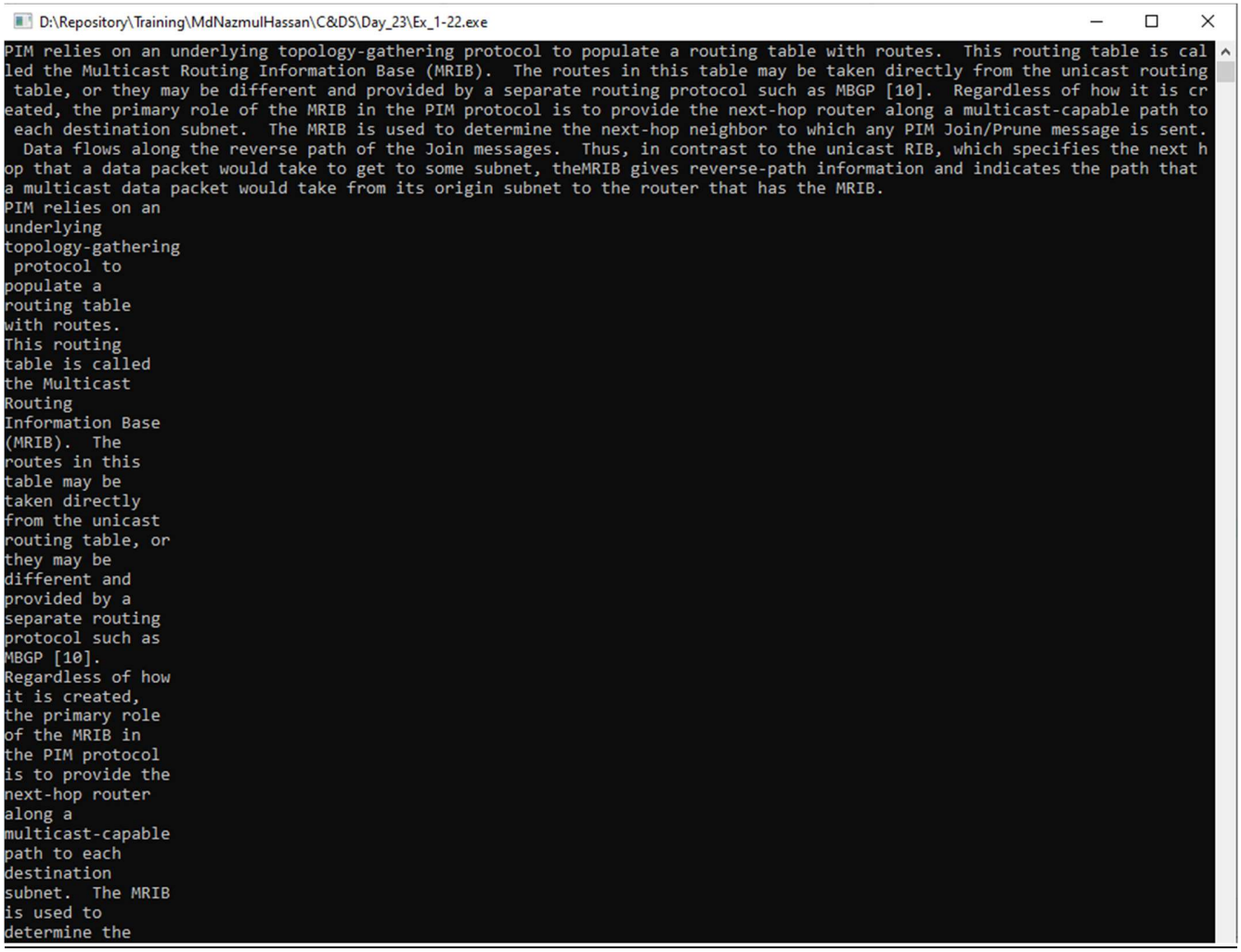
Key Points of my approach to this solution:

- The program **reads input lines** using the **get_line function**.
- The **find_longest_word function** scans the input line and **finds the length of the longest word by tracking word boundaries**.
- The **fold_line** function **iterates** through the **input line and inserts newline characters at appropriate positions to fold the line**. It keeps track of the current column and the position of the last encountered blank character to determine where to insert newline characters.
- If the **current column reaches the breaking point**, and **there is a blank character available**, it **inserts a newline character at the position of the last blank**.
- If **no blank character is available, it inserts a hyphen followed by a newline character to indicate the line continuation**.

Input:



Outputs:



- Here the **maximum word length is: 18** and the word is **topology-gathering.**
- **So the program intelligently folds long input lines based on the length of the longest word(topology-gathering) and ensures that words are not split and lines are properly wrapped.**