

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:

```
1  #include <stdio.h>
2
3  #define TAB_WIDTH 4 // Set the tab stop value as a symbolic parameter
4
5  int main(void) {
6      int c;
7      int position = 0;
8
9      while ((c = getchar()) != EOF) {
10         if (c == '\t') {
11             int spaces = TAB_WIDTH - (position % TAB_WIDTH);
12             int i;
13             for (i = 0; i < spaces; i++) {
14                 putchar('*'); /*Determines that the tabs are replaced by tabs*/
15                 position++;
16             }
17         } else if (c == '\n') {
18             putchar(c);
19             position = 0;
20         } else {
21             putchar(c);
22             position++;
23         }
24     }
25
26     return 0;
27 }
28
```

Key Points of my approach to this solution:

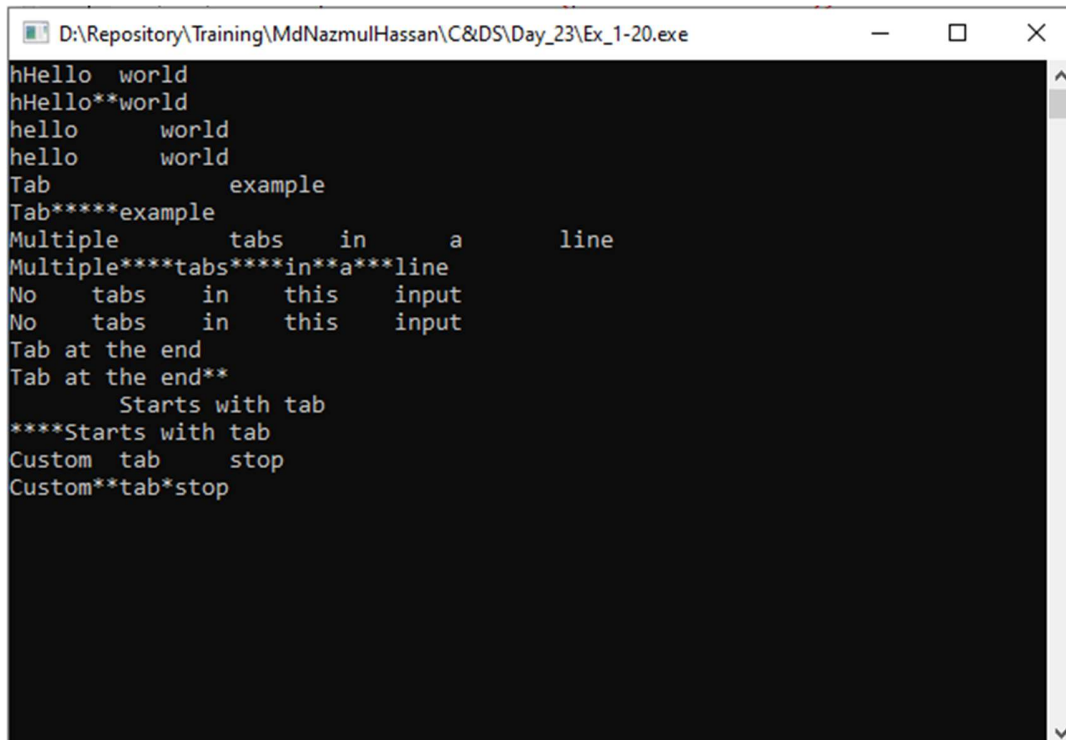
The `detab` program replaces tab characters with spaces based on a symbolic parameter, `TAB_WIDTH`. Here are the key features:

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:



```
D:\Repository\Training\MdNazmulHassan\C&DS\Day_23\Ex_1-20.exe
hHello world
hHello**world
hello world
hello world
Tab example
Tab*****example
Multiple tabs in a line
Multiple****tabs****in**a***line
No tabs in this input
No tabs in this input
Tab at the end
Tab at the end**
    Starts with tab
****Starts with tab
Custom tab stop
Custom**tab*stop
```

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:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #define MAXLINE 10000
5
6  #define TRUE (1 == 1)
7  #define FALSE !TRUE
8
9  int get_line(char line[], int max_line_len);
10 void fold_line(char line[], char fold_str[], int n_break);
11 int find_longest_word(char line[]);
12
13 int main(void)
14 {
15     char line[MAXLINE];
16     char fold_str[MAXLINE];
17
18     while (get_line(line, MAXLINE) > 0)
19     {
20         int breaking_point = find_longest_word(line);
21         fold_line(line, fold_str, breaking_point);
22         printf("%s", fold_str);
23     }
24
25     return 0;
26 }
27
28 int get_line(char line[], int max_line_len)
29 {
30     int c, i = 0;
31
32     while (i < max_line_len - 1 && (c = getchar()) != EOF && c != '\n')
33     {
34         line[i++] = c;
35     }
36
37     if (c == '\n')
38     {
39         line[i++] = c;
40     }
41
42     line[i] = '\0';
43
44     return i;
45 }
46
```

```

47 void fold_line(char line[], char fold_str[], int n_break)
48 {
49     int i, j;
50     int column = 0;
51     int last_blank = -1;
52
53     for (i = 0, j = 0; line[i] != '\0'; ++i, ++j)
54     {
55         fold_str[j] = line[i];
56
57         if (fold_str[j] == '\n')
58         {
59             column = 0;
60             last_blank = -1;
61         }
62
63         column++;
64
65         if (fold_str[j] == ' ' || fold_str[j] == '\t')
66             last_blank = j;
67
68         if (column == n_break)
69         {
70             if (last_blank != -1)
71             {
72                 fold_str[last_blank] = '\n';
73                 column = j - last_blank;
74                 last_blank = -1;
75             }
76             else
77             {
78                 fold_str[j++] = '-';
79                 fold_str[j] = '\n';
80
81                 column = 0;
82             }
83         }
84     }
85
86     fold_str[j] = '\0';
87 }
88

```

```

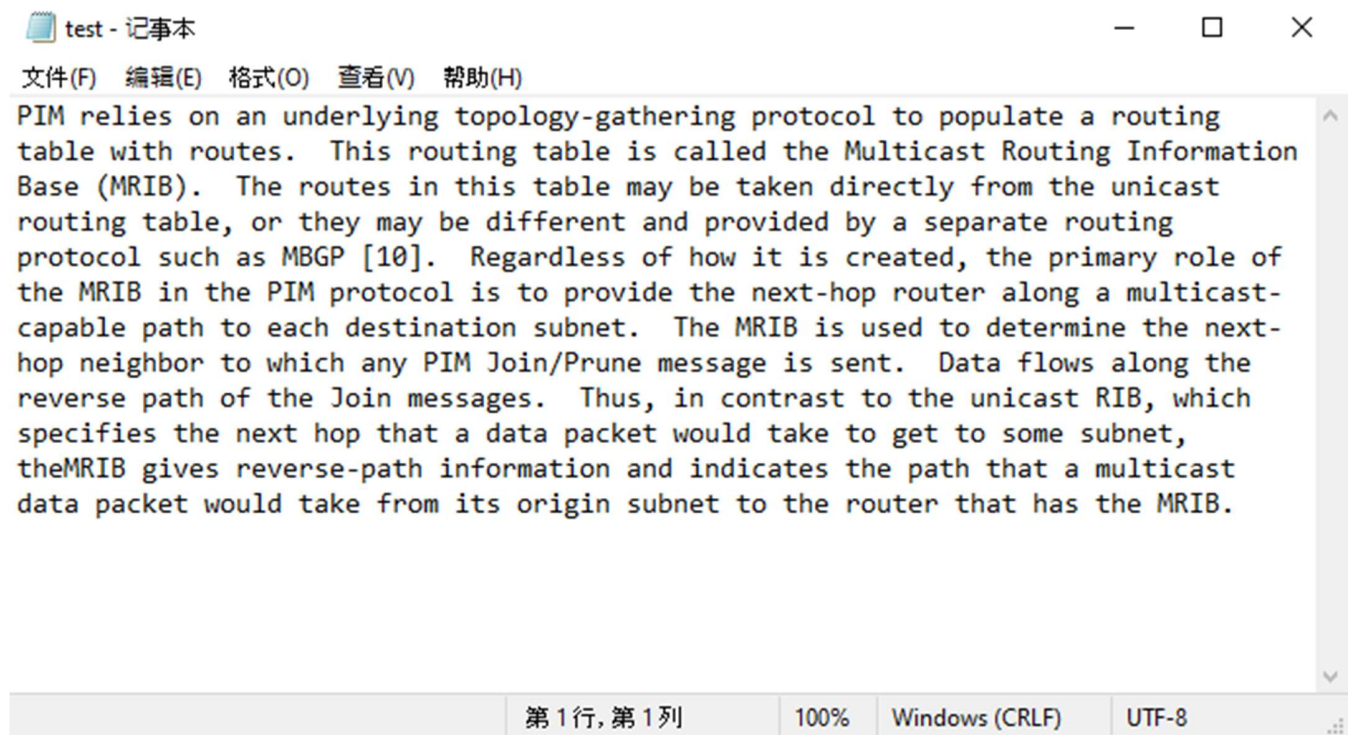
88
89 int find_longest_word(char line[])
90 {
91     int longest_word_length = 0;
92     int current_word_length = 0;
93     int i;
94
95     for (i = 0; line[i] != '\0'; i++)
96     {
97         if (line[i] == ' ' || line[i] == '\t' || line[i] == '\n')
98         {
99             if (current_word_length > longest_word_length)
100             {
101                 longest_word_length = current_word_length;
102             }
103             current_word_length = 0;
104         }
105         else
106         {
107             current_word_length++;
108         }
109     }
110
111     if (current_word_length > longest_word_length)
112     {
113         longest_word_length = current_word_length;
114     }
115
116     return longest_word_length;
117 }

```

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:



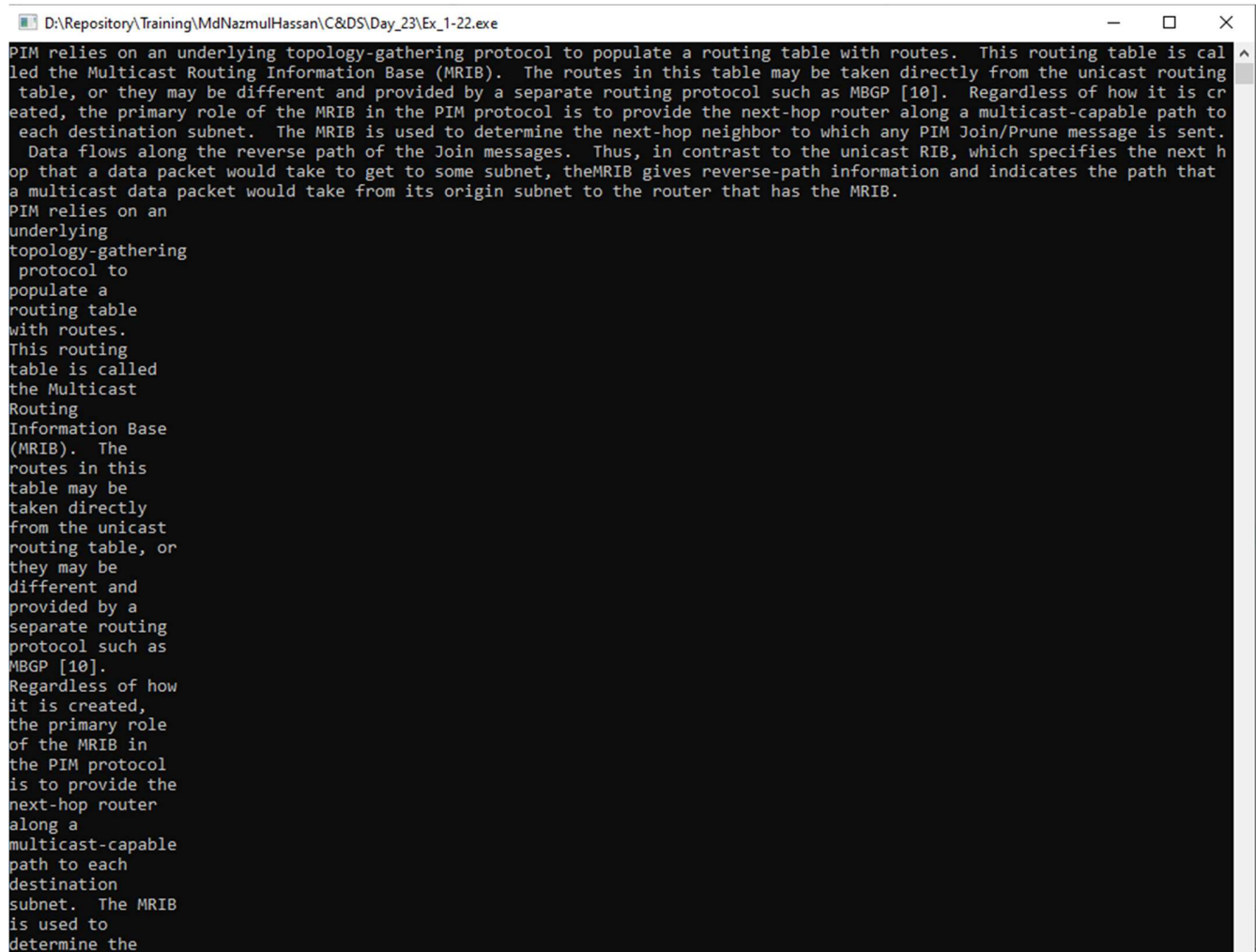
test - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

PIM relies on an underlying topology-gathering protocol to populate a routing table with routes. This routing table is called the Multicast Routing Information Base (MRIB). The routes in this table may be taken directly from the unicast routing table, or they may be different and provided by a separate routing protocol such as MBGP [10]. Regardless of how it is created, the primary role of the MRIB in the PIM protocol is to provide the next-hop router along a multicast-capable path to each destination subnet. The MRIB is used to determine the next-hop neighbor to which any PIM Join/Prune message is sent. Data flows along the reverse path of the Join messages. Thus, in contrast to the unicast RIB, which specifies the next hop that a data packet would take to get to some subnet, theMRIB gives reverse-path information and indicates the path that a multicast data packet would take from its origin subnet to the router that has the MRIB.

第 1 行, 第 1 列 100% Windows (CRLF) UTF-8

Outputs:



```
D:\Repository\Training\MdNazmulHassan\C&DS\Day_23\Ex_1-22.exe
PIM relies on an underlying topology-gathering protocol to populate a routing table with routes. This routing table is called the Multicast Routing Information Base (MRIB). The routes in this table may be taken directly from the unicast routing table, or they may be different and provided by a separate routing protocol such as MBGP [10]. Regardless of how it is created, the primary role of the MRIB in the PIM protocol is to provide the next-hop router along a multicast-capable path to each destination subnet. The MRIB is used to determine the next-hop neighbor to which any PIM Join/Prune message is sent. Data flows along the reverse path of the Join messages. Thus, in contrast to the unicast RIB, which specifies the next hop that a data packet would take to get to some subnet, the MRIB gives reverse-path information and indicates the path that a multicast data packet would take from its origin subnet to the router that has the MRIB.
PIM relies on an
underlying
topology-gathering
protocol to
populate a
routing table
with routes.
This routing
table is called
the Multicast
Routing
Information Base
(MRIB). The
routes in this
table may be
taken directly
from the unicast
routing table, or
they may be
different and
provided by a
separate routing
protocol such as
MBGP [10].
Regardless of how
it is created,
the primary role
of the MRIB in
the PIM protocol
is to provide the
next-hop router
along a
multicast-capable
path to each
destination
subnet. The MRIB
is used to
determine the
```

- 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.