

Documentation of Day_22

Exercise 7-1. Write a program that converts upper case to lower or lower case to upper, depending on the name it is invoked with, as found in `argv[0]`

Source Code:

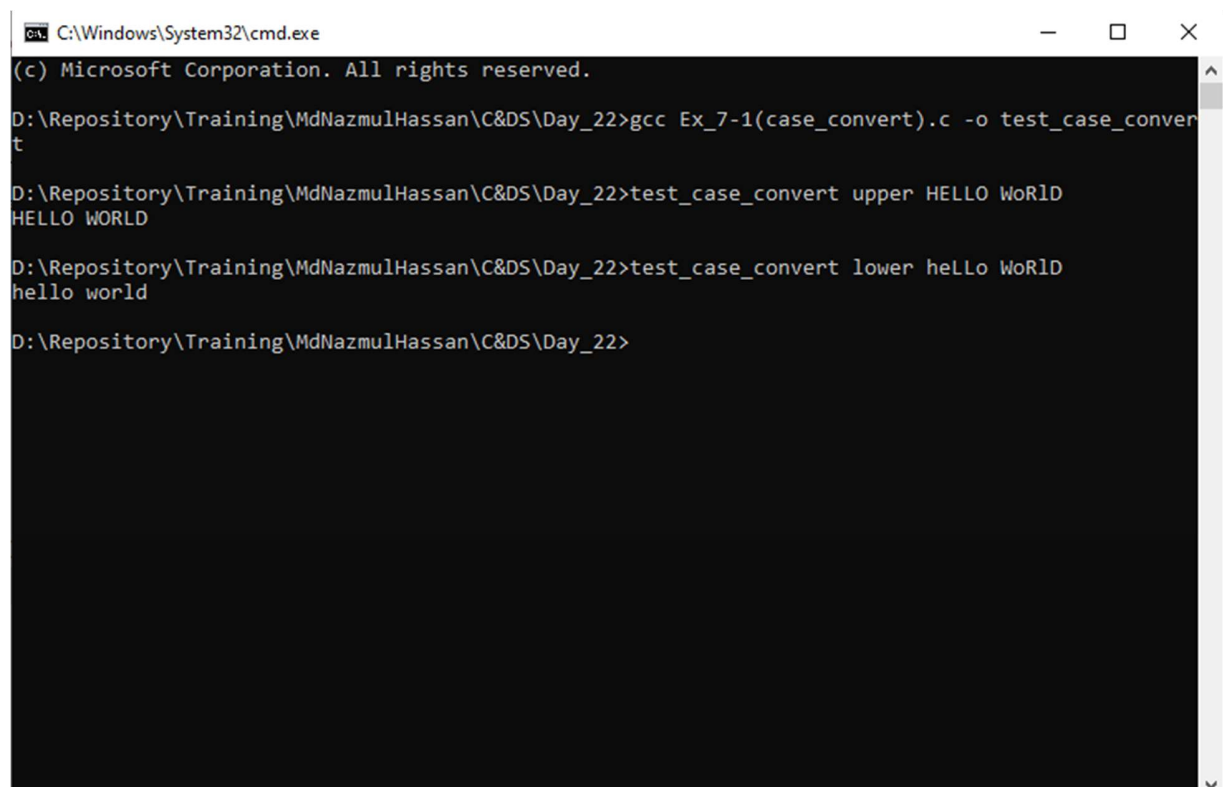
```
Ex_7-1(case_convert).c
1  #include <stdio.h>
2  #include <ctype.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6      // Check if at least one argument is provided (excluding program name)
7      if (argc < 2) {
8          printf("Usage: %s <string>\n", argv[0]);
9          return 1;
10     }
11
12     int i;
13     char* programName = argv[1];
14
15     // Determine the desired conversion based on the program name
16     int (*convertFunc)(int) = NULL;
17     if (strcmp(programName, "lower") == 0) {
18         convertFunc = tolower;
19     } else if (strcmp(programName, "upper") == 0) {
20         convertFunc = toupper;
21     }
22
23     // Convert the input string using the selected conversion function
24     if (convertFunc != NULL) {
25         for (i = 2; i < argc; i++) {
26             int j;
27             for (j = 0; argv[i][j] != '\0'; j++) {
28                 putchar(convertFunc(argv[i][j]));
29             }
30             putchar(' '); // Add a space between words
31         }
32         putchar('\n');
33     } else {
34         printf("Invalid program name: %s\n", programName);
35         return 1;
36     }
37
38     return 0;
39 }
40
41
```

Explanation:

The program aims to convert input strings to lowercase or uppercase based on the provided program name as a command-line argument. The primary focus of the code is to determine the desired conversion based on the program name. By using `strcmp` to compare the program name with the conversion options,

the code ensures flexibility for future expansion without modifying the core logic. The utilization of function pointers (convertFunc) allows for easy switching between tolower and toupper functions, enabling lowercase and uppercase conversions respectively. The design also includes error handling for invalid program names, providing informative error messages and returning a non-zero value to indicate an error. Overall, the code structure emphasizes clarity, modularity, and extensibility, allowing for straightforward maintenance and potential additions of new conversion options. By separating the "what" (determining conversion) from the "how" (conversion implementation), the code promotes code reuse and scalability.

Output:



```
C:\Windows\System32\cmd.exe
(c) Microsoft Corporation. All rights reserved.

D:\Repository\Training\MdNazmulHassan\C&DS\Day_22>gcc Ex_7-1(case_convert).c -o test_case_convert

D:\Repository\Training\MdNazmulHassan\C&DS\Day_22>test_case_convert upper HELLO WoRLD
HELLO WORLD

D:\Repository\Training\MdNazmulHassan\C&DS\Day_22>test_case_convert lower heLLo WoRLD
hello world

D:\Repository\Training\MdNazmulHassan\C&DS\Day_22>
```

Exercise 7-6. Write a program to compare two files, printing the first line where they differ.

Source Code:

```
Ex_7-6(input_file_compare).c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define MAX_LINE_LENGTH 100
6
7  int compareFiles(FILE* file1, FILE* file2);
8
9  int main() {
10     FILE* file1;
11     FILE* file2;
12
13     file1 = fopen("file1.txt", "r");
14     file2 = fopen("file2.txt", "r");
15
16     if (file1 == NULL || file2 == NULL) {
17         printf("Error opening files.\n");
18         exit(1);
19     }
20
21     int line = compareFiles(file1, file2);
22     if (line == -1) {
23         printf("Files are identical.\n");
24     } else {
25         printf("Files differ at line %d.\n", line);
26     }
27
28     fclose(file1);
29     fclose(file2);
30
31     return 0;
32 }
33
34 int compareFiles(FILE* file1, FILE* file2) {
35     char line1[MAX_LINE_LENGTH];
36     char line2[MAX_LINE_LENGTH];
37     int line = 1;
38
39     while (fgets(line1, MAX_LINE_LENGTH, file1) != NULL && fgets(line2, MAX_LINE_LENGTH, file2) != NULL) {
40         if (strcmp(line1, line2) != 0) {
41             printf("Files differ at line %d.\n", line);
42             printf("File 1: %s", line1);
43             printf("File 2: %s", line2);
44             return line;
45         }
46         line++;
47     }
48
49     // If one file has more lines than the other
50     if (fgets(line1, MAX_LINE_LENGTH, file1) != NULL) {
51         printf("File 2 ended, but File 1 still has more lines.\n");
52         printf("Remaining lines in File 1:\n");
53         printf("%s", line1);
54         return line;
55     } else if (fgets(line2, MAX_LINE_LENGTH, file2) != NULL) {
56         printf("File 1 ended, but File 2 still has more lines.\n");
57         printf("Remaining lines in File 2:\n");
58         printf("%s", line2);
59         return line;
60     }
61
62     // If both files have the same content
63     return -1;
64 }
65
```

Inputs & Output:

➔ **Input:**

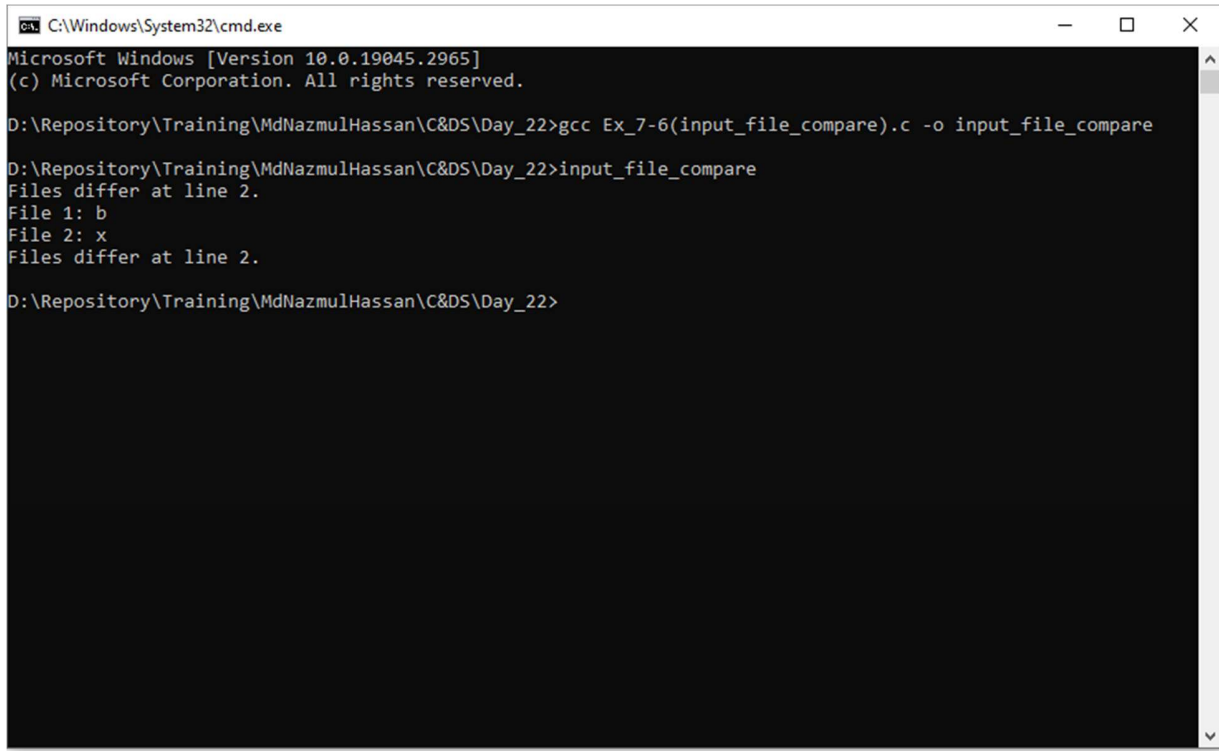
file1.txt:



file2.txt:



➔ **Output:**



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.2965]
(c) Microsoft Corporation. All rights reserved.

D:\Repository\Training\MdNazmulHassan\C&DS\Day_22>gcc Ex_7-6(input_file_compare).c -o input_file_compare

D:\Repository\Training\MdNazmulHassan\C&DS\Day_22>input_file_compare
Files differ at line 2.
File 1: b
File 2: x
Files differ at line 2.

D:\Repository\Training\MdNazmulHassan\C&DS\Day_22>
```

Explanation or Key points:

- The program aims to compare two files and identify the first line where they differ. It reads the files line by line and uses the `fgets` function to store each line in separate character arrays. The program then compares the lines using the `strcmp` function.
- To achieve the desired outcome, the program uses a while loop to iterate through the lines of both files. If a difference is found between the lines, it prints the line number and the differing lines from both files. It then returns the line number to indicate the first differing line.
- Additionally, the program handles scenarios where one file has more lines than the other. If one file ends before the other, it prints a message indicating that the remaining lines belong to the file that has not ended.
- The program uses a constant `MAX_LINE_LENGTH` to define the maximum length of a line. It opens the files using `fopen`, checks if they were opened successfully, and closes them using `fclose` at the end.
- Overall, the program reads and compares the lines of both files, identifies the first difference, and handles cases where one file has more lines.

Exercise 7-9:

Let's explore both possibilities of implementing functions like `isupper` to either save space or save time:

Saving Space:

When implementing functions to save space, the focus is on minimizing the memory footprint of the code. Here are some strategies to achieve this:

- ➔ **Bitwise Operations:** Instead of using lookup tables or arrays to store information about character properties, bitwise operations can be employed to encode the properties in a smaller space. For example, a single bit can represent whether a character is uppercase or lowercase, thus reducing memory usage.
- ➔ **Compact Data Structures:** Using compact data structures like bit arrays or bitsets can help save space when storing character properties. These structures allow efficient storage and retrieval of information using a minimal amount of memory.
- ➔ **Saving Time:** When implementing functions to save time, the emphasis is on optimizing the execution speed of the code. Here are some strategies to achieve this:
 - ➔ **Lookup Tables:** Precomputing and storing the results of expensive operations can significantly speed up function execution. For example, a lookup table can be used to store the uppercase/lowercase properties of characters, allowing for quick and direct access during runtime.
 - ➔ **Caching:** Caching frequently accessed results can help avoid redundant computations. By storing previously computed results in memory, subsequent calls to the function can be faster, as the cached values can be retrieved instead of recomputing them.
 - ➔ **Algorithmic Optimization:** Analyzing the problem and identifying algorithmic improvements can lead to substantial time savings. For instance, by using bitwise operations or logical shortcuts, certain checks can be performed more efficiently.

It's important to note that the choice between saving space or saving time depends on the specific requirements and constraints of the application. In some cases, space optimization may be more critical, such as in embedded systems with limited memory. Conversely, time optimization may be prioritized in performance-critical applications where efficient execution is paramount.

Ultimately, the decision to save space or save time when implementing functions like `isupper` depends on the trade-offs and priorities of the specific use case.

Here's an implementation of the `isupper` function in C that demonstrates both approaches: one to save space and another to save time.

Functions:

1. `isupper_space()`:

The `isupper_space` function saves space by avoiding the need for additional data structures or lookup tables to determine if a character is uppercase. Instead, it leverages the ASCII values of characters to perform the uppercase check.

In this implementation, the function checks if the ASCII value of the character `c` is within the range of uppercase characters 'A' to 'Z'. If the ASCII value falls within this range, it returns `true`, indicating that the character is uppercase. Otherwise, it returns `false`, indicating that the character is not uppercase. By directly comparing the ASCII values, this implementation eliminates the need for extra storage or complex logic. It is a concise and efficient way to determine if a character is uppercase without using additional resources.

Here is the function:

```
4 bool isupper_space(char c) {  
5     // Check if the given character is uppercase using ASCII values  
6     return c >= 'A' && c <= 'Z';  
7 }
```

2. `isupper_time()`:

The time-saving implementation utilizes a lookup table to directly access the uppercase property of each character. This approach avoids redundant computations and improves execution speed.

The `'isupper_time'` function saves time by utilizing a lookup table to directly access the uppercase property of characters without the need for any calculations or comparisons.

In this implementation, a static `const` array named `'uppercaseTable'` is used as the lookup table. The array has a size of 256, representing all possible ASCII values. By default, all elements are initialized to `'false'`.

The uppercase characters ('A' to 'Z') are explicitly marked as `'true'` in the lookup table using their corresponding ASCII values as indices. This initialization is done only once during the program's execution.

When the `'isupper_time'` function is called with a character `'c'`, it directly accesses the element at index `'c'` in the `'uppercaseTable'` array. This operation is constant-time ($O(1)$) since it involves a simple array access.

The function then returns the value at the accessed index, indicating whether the character is uppercase or not.

By using the lookup table, the `'isupper_time'` function eliminates the need for repetitive checks or calculations, resulting in a constant-time lookup operation. This approach can provide a significant time-saving advantage, especially when multiple character checks are required.

Here is the function:

```
9 bool isupper_time(char c) {  
10     // Lookup table to store the uppercase property of characters  
11     static const bool uppercaseTable[256] = {  
12         /* 0-255: false */  
13         /* ... */  
14         /* 'A'-'A' to 'Z'-'A': true */  
15         ['A'] = true, // A  
16         ['B'] = true, // B  
17         ['C'] = true, // C  
18         // ... and so on for all uppercase characters  
19     };  
20  
21     // Check if the given character is uppercase  
22     return uppercaseTable[(unsigned char)c];  
23 }
```

The `isupper_time` function saves time by utilizing a lookup table to directly access the uppercase property of characters without the need for any calculations or comparisons.

In this implementation, a static const array named `uppercaseTable` is used as the lookup table. The array has a size of 256, representing all possible ASCII values. By default, all elements are initialized to false.

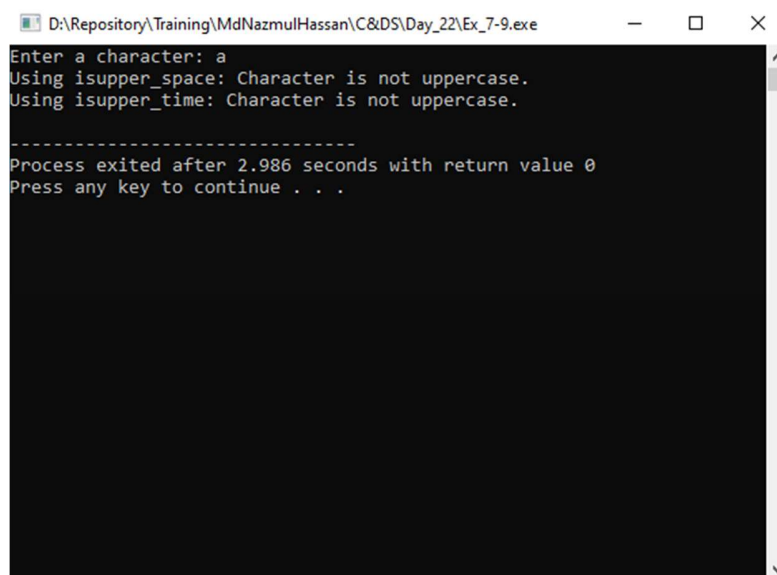
The uppercase characters ('A' to 'Z') are explicitly marked as true in the lookup table using their corresponding ASCII values as indices. This initialization is done only once during the program's execution.

When the `isupper_time` function is called with a character `c`, it directly accesses the element at index `c` in the `uppercaseTable` array. This operation is constant-time ($O(1)$) since it involves a simple array access.

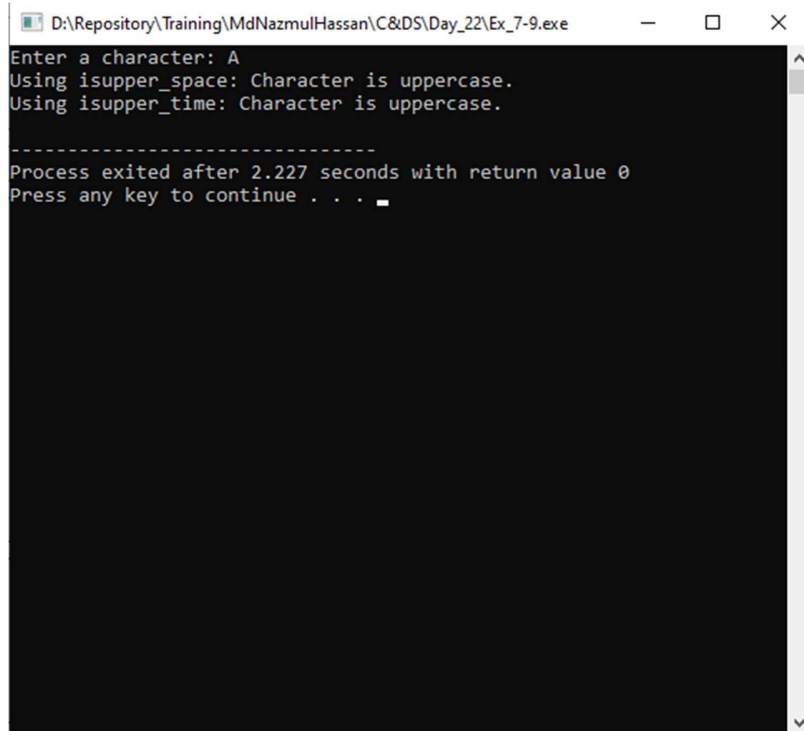
The function then returns the value at the accessed index, indicating whether the character is uppercase or not.

By using the lookup table, the `isupper_time` function eliminates the need for repetitive checks or calculations, resulting in a constant-time lookup operation. This approach can provide a significant time-saving advantage, especially when multiple character checks are required.

Outputs:



```
D:\Repository\Training\MdNazmulHassan\C&DS\Day_22\Ex_7-9.exe  
Enter a character: a  
Using isupper_space: Character is not uppercase.  
Using isupper_time: Character is not uppercase.  
-----  
Process exited after 2.986 seconds with return value 0  
Press any key to continue . . .
```

```
D:\Repository\Training\MdNazmulHassan\C&DS\Day_22\Ex_7-9.exe
Enter a character: A
Using isupper_space: Character is uppercase.
Using isupper_time: Character is uppercase.
-----
Process exited after 2.227 seconds with return value 0
Press any key to continue . . .
```