East West University

Department of Computer Science and Engineering

Submitted to: Dr. Md. Nawab Yousuf Ali
Professor, Department of Computer Science and Engineering

Submitted by:

Name	**ID**	**Contribution**	**Roll no**
Tamim Rafi Ahmed	2021-3-60-089	25%	6
Gazi Md Jubayer	2022-2-60-080	25%	21
Md. Sadik sabbir	2022-1-60-020	25%	8
Aklhak Hossain	2022-3-60-057	25%	23

Course Title: Computer Architecture
Course code: cse360
Semester: Fall 2025
Section: 1
Group: 10
Project Name: Design a Turing machine using C to count character occurrences
Date: 19 May, 2025

Objective

The objective of this project is to design and implement a Machine simulator in C that counts the number of occurrences of a specified character in a given input string. The program simulates the behavior of a single-tape deterministic Turing Machine by explicitly modeling the tape, tape head, and finite control states.

The Turing Machine reads the input string placed on the tape, scans symbols sequentially, identifies occurrences of a target character, and increments a counter whenever a match is found. The tape head moves in a single direction, and matched characters are marked to avoid reprocessing. The machine halts upon reaching a blank symbol, at which point the total count is displayed.

By implementing this functionality, the program demonstrates how abstract computational models such as Turing Machines can be represented and executed using a high-level programming language like C.

Theory

A Turing Machine is a theoretical model of computation introduced by Alan Turing to define the concept of algorithmic processing. It consists of an infinite tape divided into cells, a tape head that can read and write symbols, and a finite set of states that govern the machine’s behavior through transition rules.

In this implementation, the Turing Machine is simulated using a finite-length array to represent the tape, an integer index to represent the tape head position, and an enumeration to represent the machine’s states. The machine operates by repeatedly reading the symbol under the tape head, transitioning between states based on predefined rules, optionally writing symbols to the tape, and moving the tape head.

The execution follows a step-by-step state transition process similar to the theoretical operation of a Turing Machine. Although the simulator runs on a conventional computer, it faithfully models the logical behavior of a Turing Machine, demonstrating how fundamental computational concepts can be implemented and observed in practice.

Instruction Semantics

The behavior of the Turing Machine is defined by a set of states and transitions that operate on the tape and control head movement. Each transition determines how the machine reads symbols, updates the tape, and moves between states.

Instructions	Semantics
Q_START	Initial state of the Turing Machine. The machine initializes execution and immediately transitions to the scanning state.
Q_SCAN	The tape head reads the current symbol. If the symbol is blank, the machine halts and enters the accept state. If the symbol matches the target character, the machine transitions to the found state. Otherwise, the tape head moves one position to the right and continues scanning.
Q_FOUND	A matching character has been detected. The occurrence counter is incremented, the current tape symbol is replaced with a marker character to prevent re-counting, and the machine transitions to the continue state.
Q_CONTINUE	The tape head moves one position to the right and control returns to the scanning state.
Q_ACCEPT	Final accepting state of the machine.

Design

The Turing Machine simulator is structured around initialization and execution phases.

Rather than parsing an external instruction file, the machine's behavior is defined by a

finite set of states and transition rules encoded directly in the program. The overall design

focuses on simulating tape-based computation through controlled state transitions.

The design steps are described below:

1. Initialization Phase

- Read the input string from standard input.
- Initialize the tape by copying the input string into a fixed-size tape array and filling remaining cells with a blank symbol.
- Set the tape head position to the beginning of the tape.
- Initialize the Turing Machine's current state to the start state.
- Store the target character to be counted and initialize the counter to zero.
- This phase prepares the internal representation of the Turing Machine before execution begins.

2. Execution Phase (State Transition Loop)

- The simulation follows a loop that continues until the machine reaches the accept state.
- Read the symbol currently under the tape head.
- Based on the current state and the read symbol, determine the next action.

Perform one or more of the following operations:

- Transition to a new state
- Modify the tape symbol (mark matched characters)
- Move the tape head to the right
- Update the occurrence counter
- The execution proceeds sequentially, closely mirroring the theoretical operation of a deterministic Turing Machine.

3. State Transition Logic

The machine behavior is governed by a finite set of states:

o **Q_START:** Initializes execution and transitions to the scanning state.

o **Q_SCAN:** Reads symbols from the tape one by one.

o **Q_FOUND**: Handles a successful match by incrementing the counter and marking the tape.

o **Q_CONTINUE**: Moves the tape head forward and resumes scanning.

o **Q_ACCEPT**: Final halting state indicating completion of computation.

Transitions between these states are implemented using a switch-case control structure.

4. Data Structures and Memory

o A fixed-size character array tape[LENGTH] simulates the Turing Machine tape.

o An integer variable tracks the tape head position.

o An enumeration type represents the current state of the machine.

o A structure TuringMachine encapsulates the tape, head position, current state, target character, and occurrence counter.

This design ensures that all components of the Turing Machine are maintained in a single cohesive structure.

Instruction (State) Effect Summary

State	Effect on machine state
Q_START	Initialize execution and move to scan state
Q_SCAN	Read tape symbol and decide next transition
Q_FOUND	Increment count and mark matched symbol
Q_CONTINUE	Move tape head right
Q_ACCEPT	Stop execution

This design demonstrates how a theoretical Turing Machine can be mapped to concrete data structures and control flow in C, allowing abstract computation models to be simulated on conventional hardware.

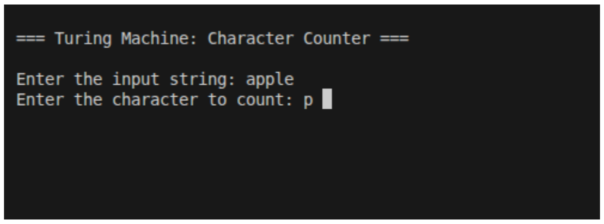
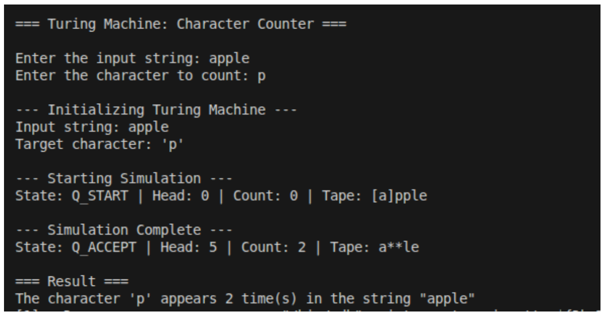
Source Code

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

#define LENGTH 1000
#define BLANK '_'

typedef enum
{
    Q_START,
    Q_SCAN,
    Q_FOUND,
    Q_CONTINUE,
    Q_ACCEPT
} State;

typedef struct
{
    char tape[LENGTH];
    int head_position;
    State current_state;
    char target_char;
    int count;
} TuringMachine;

void init_turing_machine(TuringMachine *tm, const char *input, char target)
{
    int len = strlen(input);

    for (int i = 0; i < LENGTH; i++)
    {
        if (i < len)
        {
            tm->tape[i] = input[i];
        }
        else
        {
            tm->tape[i] = BLANK;
        }
    }

    tm->head_position = 0;
    tm->current_state = Q_START;
    tm->target_char = target;
    tm->count = 0;
}

void print_tm_state(TuringMachine *tm, int show_tape)
{
    printf("State: ");
    switch (tm->current_state)
    {
    case Q_START:
        printf("Q_START");
        break;
    case Q_SCAN:
        printf("Q_SCAN");
        break;
    case Q_FOUND:
        printf("Q_FOUND");
        break;
    case Q_CONTINUE:
        printf("Q_CONTINUE");
        break;
    case Q_ACCEPT:
        printf("Q_ACCEPT");
        break;
    }
    printf(" | Head: %d | Count: %d", tm->head_position, tm->count);

    if (show_tape)
    {
        printf(" | Tape: ");
        for (int i = 0; i < 50 && tm->tape[i] != BLANK; i++)
        {
            if (i == tm->head_position)
            {
                printf("[%c]", tm->tape[i]);
            }
            else
            {
                printf("%c", tm->tape[i]);
            }
        }
    }
    printf("\n");
}

void simulate_turing_machine(TuringMachine *tm)
{
    char current_symbol;

    while (tm->current_state != Q_ACCEPT)
    {
        current_symbol = tm->tape[tm->head_position];

        switch (tm->current_state)
        {
        case Q_START:
            tm->current_state = Q_SCAN;
            break;

        case Q_SCAN:
            if (current_symbol == BLANK)
            {
                tm->current_state = Q_ACCEPT;
            }
            else if (current_symbol == tm->target_char)
            {
                tm->current_state = Q_FOUND;
            }
            else
            {
                tm->head_position++;
            }
            break;

        case Q_FOUND:
            tm->count++;
            tm->tape[tm->head_position] = '*';
            tm->current_state = Q_CONTINUE;
            break;

        case Q_CONTINUE:
            tm->head_position++;
            tm->current_state = Q_SCAN;
            break;

        case Q_ACCEPT:
            break;
        }
    }
}

int main()
{
    char input_string[LENGTH];
    char target_char;
    TuringMachine tm;

    printf("=== Turing Machine: Character Counter ===\n\n");

    printf("Enter the input string: ");
    if (fgets(input_string, sizeof(input_string), stdin) == NULL)
    {
        printf("Error reading input string.\n");
        return 1;
    }

    int len = strlen(input_string);
    if (len > 0 && input_string[len - 1] == '\n')
    {
        input_string[len - 1] = '\0';
    }

    printf("Enter the character to count: ");
    if (scanf(" %c", &target_char) != 1)
    {
        printf("Error reading target character.\n");
        return 1;
    }

    printf("\n--- Initializing Turing Machine ---\n");
    printf("Input string: %s\n", input_string);
    printf("Target character: '%c'\n\n", target_char);

    init_turing_machine(&tm, input_string, target_char);

    printf("--- Starting Simulation ---\n");
    print_tm_state(&tm, 1);
    simulate_turing_machine(&tm);

    printf("\n--- Simulation Complete ---\n");
    print_tm_state(&tm, 1);

    printf("\n=== Result ===\n");
    printf("The character '%c' appears %d time(s) in the string \"%s\"\n",
           target_char, tm.count, input_string);

    return 0;
}
```

Implementation

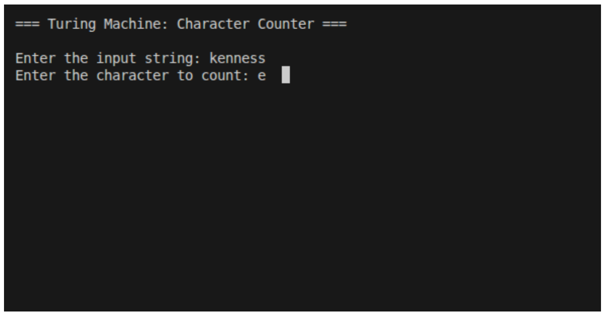
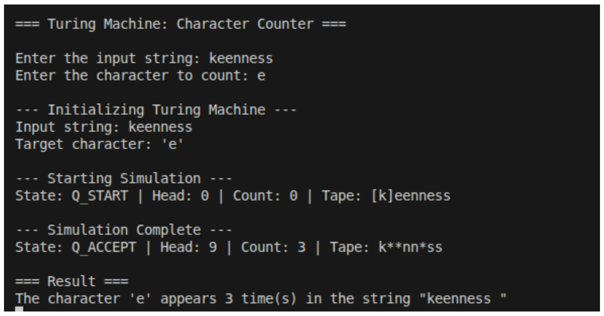
• Language: C
• Compiler: GCC compiler
• Operating System: Linux, Windows
Modules:
o Tape memory setup and initialization
o State transition and execution logic
o Head movement and symbol processing
o Debug and state output
The Turing Machine simulator was implemented in C using standard libraries such as stdio.h, stdlib.h, and string.h. The program models a single-tape deterministic Turing Machine that scans an input string and counts occurrences of a specified character.

The code is organized into well-defined functions for initialization, simulation, and debug output. The key data structures used in the implementation are:
• An enum State to represent the finite control states of the Turing Machine (Q_START, Q_SCAN, Q_FOUND, Q_CONTINUE, Q_ACCEPT).
• A struct TuringMachine that encapsulates the tape array, head position, current state, target character, and occurrence counter.
• A fixed-size character array tape[LENGTH] to simulate the Turing Machine tape, with blank symbols represented using a predefined constant.
The simulation proceeds by repeatedly reading the symbol under the tape head, performing state transitions according to the current state and input symbol, updating the tape when a target character is found, and moving the head to the right. The machine halts when it reaches the accept state.
The program was compiled using the GCC compiler and tested on both Linux and Windows platforms. Debug output is provided through a dedicated function that displays the current state, head position, count value, and a portion of the tape during execution.

Input	**Output**
	

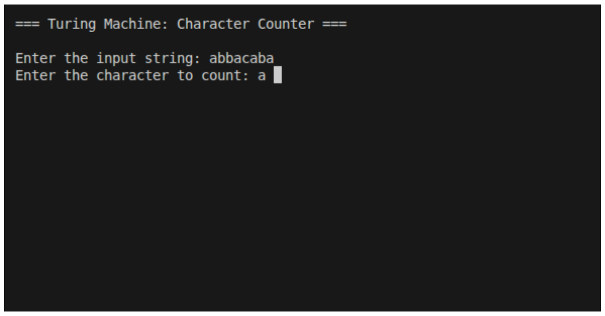
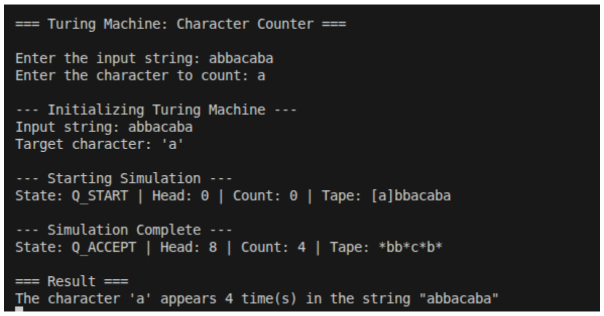
```
=== Turing Machine: Character Counter ===

Enter the input string: apple
Enter the character to count: p
```

```
=== Turing Machine: Character Counter ===

Enter the input string: apple
Enter the character to count: p

--- Initializing Turing Machine ---
Input string: apple
Target character: 'p'

--- Starting Simulation ---
State: Q_START | Head: 0 | Count: 0 | Tape: [a]pple

--- Simulation Complete ---
State: Q_ACCEPT | Head: 5 | Count: 2 | Tape: a**le

=== Result ===
The character 'p' appears 2 time(s) in the string "apple"
```

Input	Output
	

```
=== Turing Machine: Character Counter ===

Enter the input string: kenness
Enter the character to count: e
```

```
=== Turing Machine: Character Counter ===

Enter the input string: keenness
Enter the character to count: e

--- Initializing Turing Machine ---
Input string: keenness
Target character: 'e'

--- Starting Simulation ---
State: Q_START | Head: 0 | Count: 0 | Tape: [k]eenness

--- Simulation Complete ---
State: Q_ACCEPT | Head: 9 | Count: 3 | Tape: k**nn*ss

=== Result ===
The character 'e' appears 3 time(s) in the string "keenness "
```

Input	Output
	

```
=== Turing Machine: Character Counter ===

Enter the input string: abbacaba
Enter the character to count: a
```

```
=== Turing Machine: Character Counter ===

Enter the input string: abbacaba
Enter the character to count: a

--- Initializing Turing Machine ---
Input string: abbacaba
Target character: 'a'

--- Starting Simulation ---
State: Q_START | Head: 0 | Count: 0 | Tape: [a]bbacaba

--- Simulation Complete ---
State: Q_ACCEPT | Head: 8 | Count: 4 | Tape: *bb*c*b*

=== Result ===
The character 'a' appears 4 time(s) in the string "abbacaba"
```

Debugging and Test Run

We used multiple sample input strings to verify that the Turing Machine simulator functions correctly. Test cases included strings with zero occurrences of the target character, multiple occurrences, consecutive matches, and mixed characters. Edge cases such as empty strings and single-character inputs were also tested.

For debugging purposes, printf statements were inserted to trace execution step by step. After each state transition, the program printed the current state, tape head position, counter value, and a snapshot of the tape. This helped verify that state transitions, head movement, and tape updates were occurring as expected.

One of the main issues encountered during debugging was improper head advancement, which caused the machine to reprocess characters or skip symbols. Another issue involved incorrect handling of the blank symbol at the end of the tape, which initially prevented the machine from reaching the accept state. These bugs were identified and resolved using detailed execution traces printed during simulation.

Sample Test Run

Consider the following input:

Input string:
abbacaba
Target character:
a
Initial tape configuration:
abbacaba________

Expected execution behavior:

- The machine scans the tape from left to right.
- Each time the symbol a is encountered:
  - The counter is incremented.
  - The symbol is replaced with * to mark it as processed.
- The machine halts upon reaching the blank symbol.

Expected step-by-step behavior:

- First a found → count = 1
- Second a found → count = 2
- Third a found → count = 3
- Fourth a found → count = 4

Final tape (partial):
*bb*c*b*

Observed Output

From the console output:

- Final state: Q_ACCEPT
- Counter value at halt: 4
- Tape correctly marked at all matched positions

The observed output exactly matched the expected behavior. The Turing Machine correctly identified each occurrence of the target character, updated the tape to avoid re-counting, and halted at the appropriate time. This confirms that the state transitions, head movement logic, and the counting mechanism was implemented correctly.

Results Analysis

Performance:
The Turing Machine simulator operates in linear time with respect to the length of the input string. Each tape cell is scanned exactly once as the tape head moves from left to right without backtracking. Therefore, the time complexity is O(N), where *N* is the number of characters in the input string. The space complexity is also O(N) due to the fixed-size tape array used to store the input and blank symbols. Since the tape length is bounded by a predefined constant, both time and space usage remain small and predictable in practice.

Robustness:
The program handles valid inputs reliably. It correctly initializes the tape, processes each symbol according to the defined state transitions, and halts when a blank symbol is reached. The simulator avoids infinite loops by ensuring that the tape head always advances forward. The use of clearly defined states prevents undefined transitions. However, the program does not explicitly check for tape boundary overflow; extremely long input strings beyond the tape size could cause undefined behavior. This limitation is documented as a user responsibility.

Edge Cases:
• Empty Input String: The machine immediately encounters a blank symbol and halts, producing a count of zero.
• No Target Character Present: The tape is scanned fully without entering the found state, resulting in a count of zero.
• All Characters Match: Each symbol is counted and marked correctly without re-counting.
• Single Character Input: The machine correctly handles minimal input size.
• Case Sensitivity: Character comparison is case-sensitive; for example, 'a' and 'A' are treated as different symbols.

Overall, the simulator performed correctly in all tested scenarios. The observed behavior matched the expected theoretical operation of a deterministic single-tape Turing Machine.

Conclusion and Future Improvements

In this project, we successfully designed and implemented a Turing Machine simulator in C to count the occurrences of a specific character in a string. The project demonstrated how abstract models of computation can be translated into concrete implementations using data structures and control logic in a high-level programming language. The design clearly modeled the tape, tape head, finite states, and state transitions of a theoretical Turing Machine.

The implementation validated core concepts from the theory of computation, such as sequential symbol processing, halting conditions, and state-based control flow. Debugging output further helped verify correctness and provided insight into the machine's internal operation.

Limitations:
The current simulator is intentionally simple. It uses a fixed-size tape, supports movement in only one direction, and implements a single-purpose machine for character counting. There is no dynamic tape expansion, leftward head movement, or generalized transition table. Input validation and tape bounds checking are minimal.

Future Enhancements:
Several extensions could improve the simulator:
- Support for bidirectional tape head movement
- Dynamic tape resizing to better reflect an infinite tape
- A generalized transition table to simulate arbitrary Turing Machines
- Step-by-step execution with user-controlled stepping
- File-based input instead of standard input
- Enhanced error handling and input validation

These enhancements would make the simulator more flexible and closer to a fully general Turing Machine while maintaining its educational value.

Bibliography

Hopcroft, J. E., Motwani, R., & Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education.

Sipser, M. *Introduction to the Theory of Computation*. Cengage Learning.

Lewis, H. R., & Papadimitriou, C. H. *Elements of the Theory of Computation*. Prentice Hall.

Linz, P. *An Introduction to Formal Languages and Automata*. Jones & Bartlett Learning.

Kernighan, B. W., & Ritchie, D. M. *The C Programming Language*. Prentice Hall.

Tanenbaum, A. S., & Austin, T. *Structured Computer Organization*. Pearson.

Stallings, W. *Computer Organization and Architecture*. Pearson Education.

GeeksforGeeks. “Turing Machine Introduction.”
https://www.geeksforgeeks.org/turing-machine-introduction

TutorialsPoint. “Theory of Computation – Turing Machines.”
https://www.tutorialspoint.com/automata_theory/turing_machine_introduction.htm

ISO/IEC 9899:2018 – *Programming Languages — C*.