Assiut University
Faculty of Computer and Information
Department Computer Science

Deterministic Finite Automata (DFA)

Team Members:

1. Enas Ikram Girgis	CS.1
2. Aya Abdellatif Mohamed	CS.2
3. Jessica Ayman Naeam	CS.2
4. Moreen Mohsen Wagheem	CS.5

Table of contents

1. Introduction

-First, Let's talk about Automata theory briefly and DFA machines .

1.1 Automata Theory:

- **Definition:**

The word ***Automata*** comes from the Greek word **αὐτόματος**, which means "self-acting, self-willed, self-moving". **An automaton** (automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically. An automaton with a finite number of states is called a Finite Automaton (FA) or Finite-State Machine (FSM). [1]

Theory of automata is a theoretical branch of computer science and mathematical. It is the study of abstract machines and the computation problems that can be solved using these machines. The abstract machine is called the automata. The main motivation behind developing the automata theory was to develop methods to describe and analyze the dynamic behavior of discrete systems [2]

- **Objectives**

The **major objective** of automata theory is to develop methods by which computer scientists can describe and analyze the dynamic behavior of discrete systems, in which signals are sampled periodically. The behavior of these discrete systems is determined by the way that the system is constructed from storage and combinational elements.

- **Characteristics**

-Inputs: assumed to be sequences of symbols selected from a finite set *I* of input signals. Namely, set *I* is the set {x_1, $x_{,2}$, x_3… x_k} where *k* is the number of inputs.

-Outputs: sequences of symbols selected from a finite set Z. Namely, set *Z* is the set {y_1, y_2, y_3 ... y_m} where *m* is the number of outputs.

-States: finite set *Q*, whose definition depends on the type of automaton. [3]

- **Types**

There are **four major families of automaton**:

1) Finite-state machine
2) Pushdown automata
3) Linear-bounded automata
4) Turing machine

1.2 Deterministic Finite Automata (DFA)

- Deterministic finite automata (or DFA) are finite state machines that accept or reject strings of characters by parsing them through a sequence that is uniquely determined by each string.

- The term "deterministic" refers to the fact that each string, and thus each state sequence, is unique. In a DFA, a string of symbols is parsed through a DFA automata, and each input symbol will move to the next state that can be determined.

- These machines are called finite because there are a limited number of possible states which can be reached. A finite automaton is only called deterministic if it can fulfill both conditions. DFAs differ from non-deterministic automata in that one letter can transition to more than one state at a time and be active in multiple states at once. In practice, DFAs are made up of five components (and they're often denoted by a five-symbol set known as a 5-tuple). [4]

2. Machine Design

2.1 Formal Definition

A deterministic finite automaton (DFA) is a 5-tuple (Q, Σ, δ, q0, F), where:

- **Q:** is a finite set called **the states**,
- **Σ:** is a finite set called **the alphabet**,
- **δ(Q × Σ → Q)** is the **transition** function,
- **q0** ∈ **Q:** is the **start state**, and
- **F** ⊆ **Q:** is the **set of accept states**.

2.2 Case Study

2.2.1 DFA Machine1:

Machine 1 is accepting any string that has even 0's and even 1's. [4]

By using formal definition:
- Q = {q0, q1, q2, q3}.
- Σ = {0, 1}.
- q0 = {q0}.
- F = {q0}.
- δ: From transition function **Q × Σ → Q**

	0	**1**
Q0	Q2	Q1
Q1	Q3	Q0
Q2	Q0	Q3
Q3	Q1	Q2

Table 2.2.1

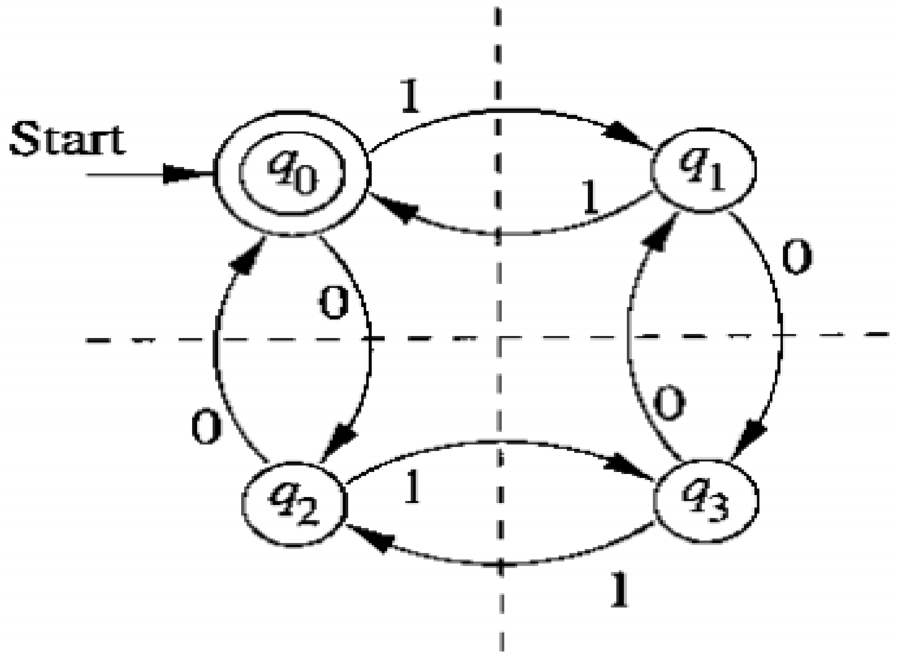


Figure 2.2.1 DFA machine1

2.2.2 DFA Machine2:

Machine 2 is accepting all the string over alphabets ∑ {0, 1} where each string contains "101" as a substring. [5]

By using formal definition:
- Q = {q0, q1, q2, q3}.
- Σ = {0, 1}.
- q0 = {q0}.
- F = {q3}.
- δ: From transition function **Q × Σ → Q**

	0	**1**
Q0	Q0	Q1
Q1	Q2	Q1
Q2	Q0	Q3
Q3	Q3	Q3

Table 2.2.2

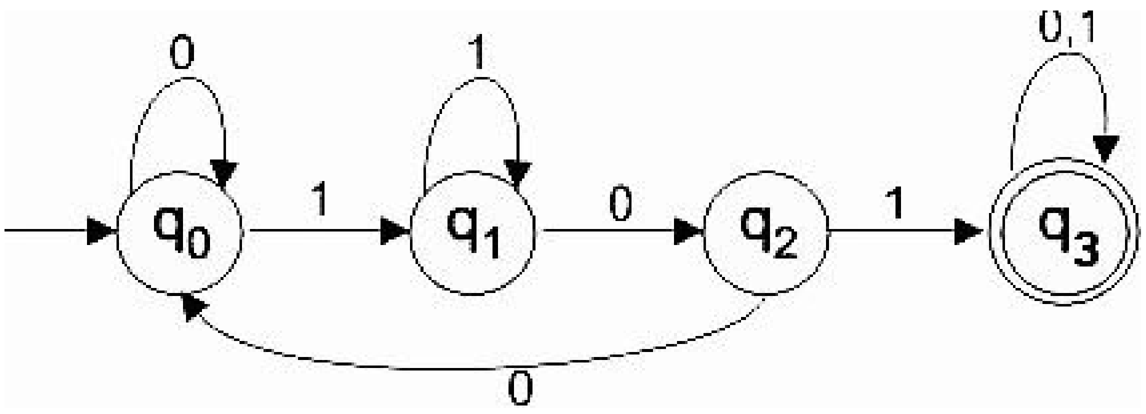


Figure 2.2.2 DFA machine2

2.2.3 DFA machine3:

Machine 3 is accepting all the string that **starts and ends with the same symbol.**[6]

By using formal definition:
- Q = {q0, q1, q2, q3, q4}.
- Σ = {a, b}.
- q0 = {q0}.
- F = {q1, q3}.
- δ: From transition function **Q × Σ → Q**

	a	b
Q0	Q1	Q3
Q1	Q1	Q2
Q2	Q1	Q2
Q3	Q4	Q3
Q4	Q4	Q3

Table 2.2.3

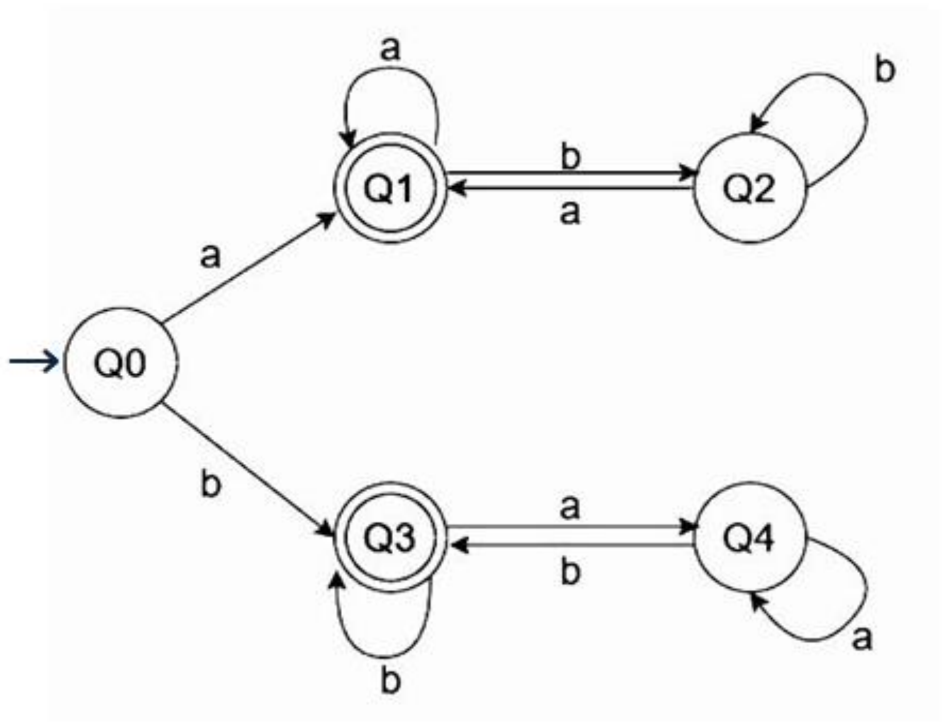


Figure 2.2.3 DFA machine 3

3. Machine Implementation

Here, We wil talk about the machine and its implementation.

3.1 Code Illustration:

Language: C++

Structure: dynamic (vectors, map)

Code link: https://ideone.com/l9JLIC

Objective: check the validation of any string of any DFA

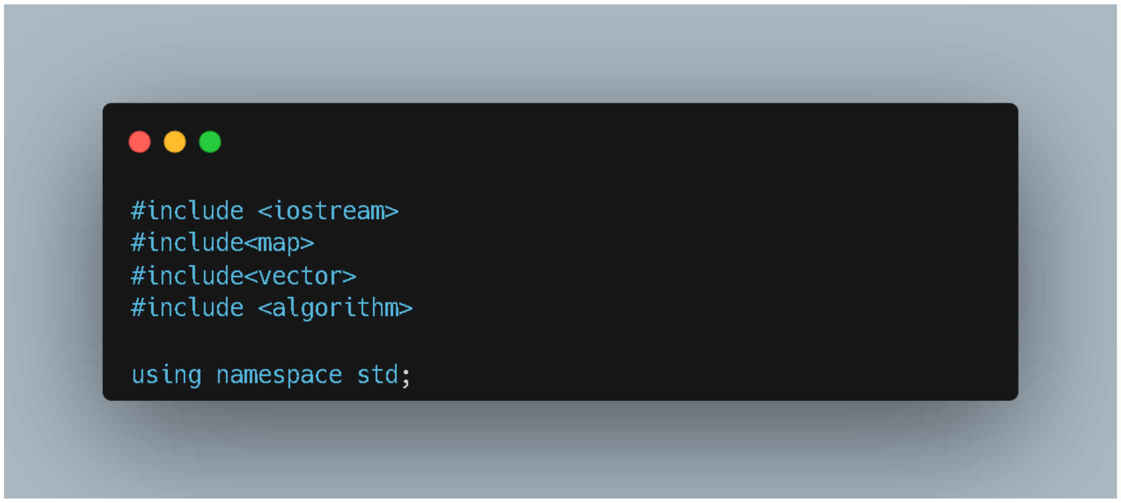
Input: all the DFA machine requirements (number of states, alphabets, start state, number of accept states, accept states, transitions of each state, strings)

Output: Accepted if the DFA machine accepts the string, Rejected if the DFA machine does not.

3.2 Project Code:

Let’s discuss every part of the code.

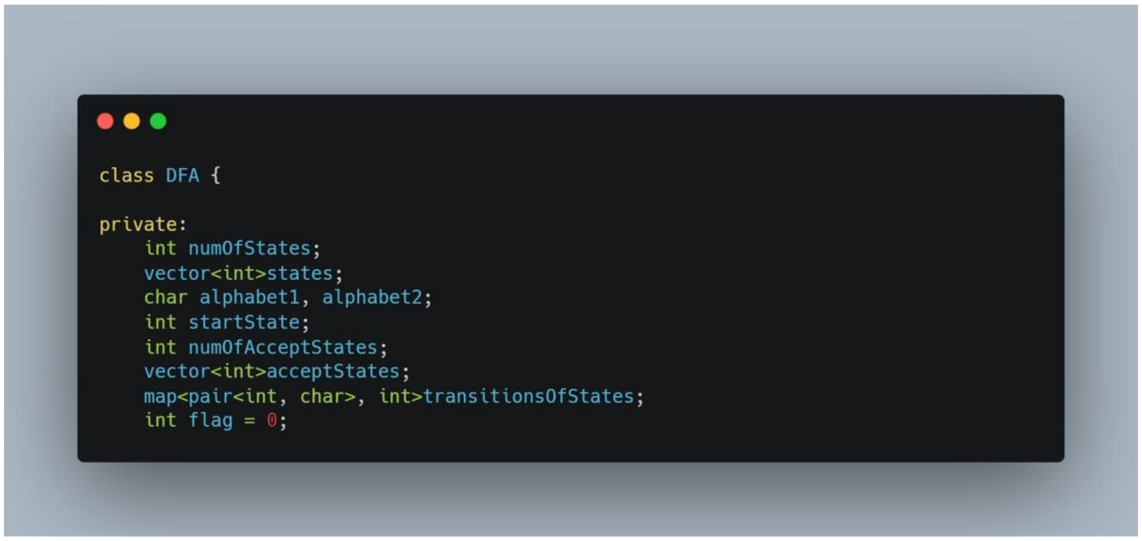
In Figure 3.2.1 we included all needed libraries for the implementation.



```
#include <iostream>
#include<map>
#include<vector>
#include <algorithm>

using namespace std;
```

Figure 3.2.1

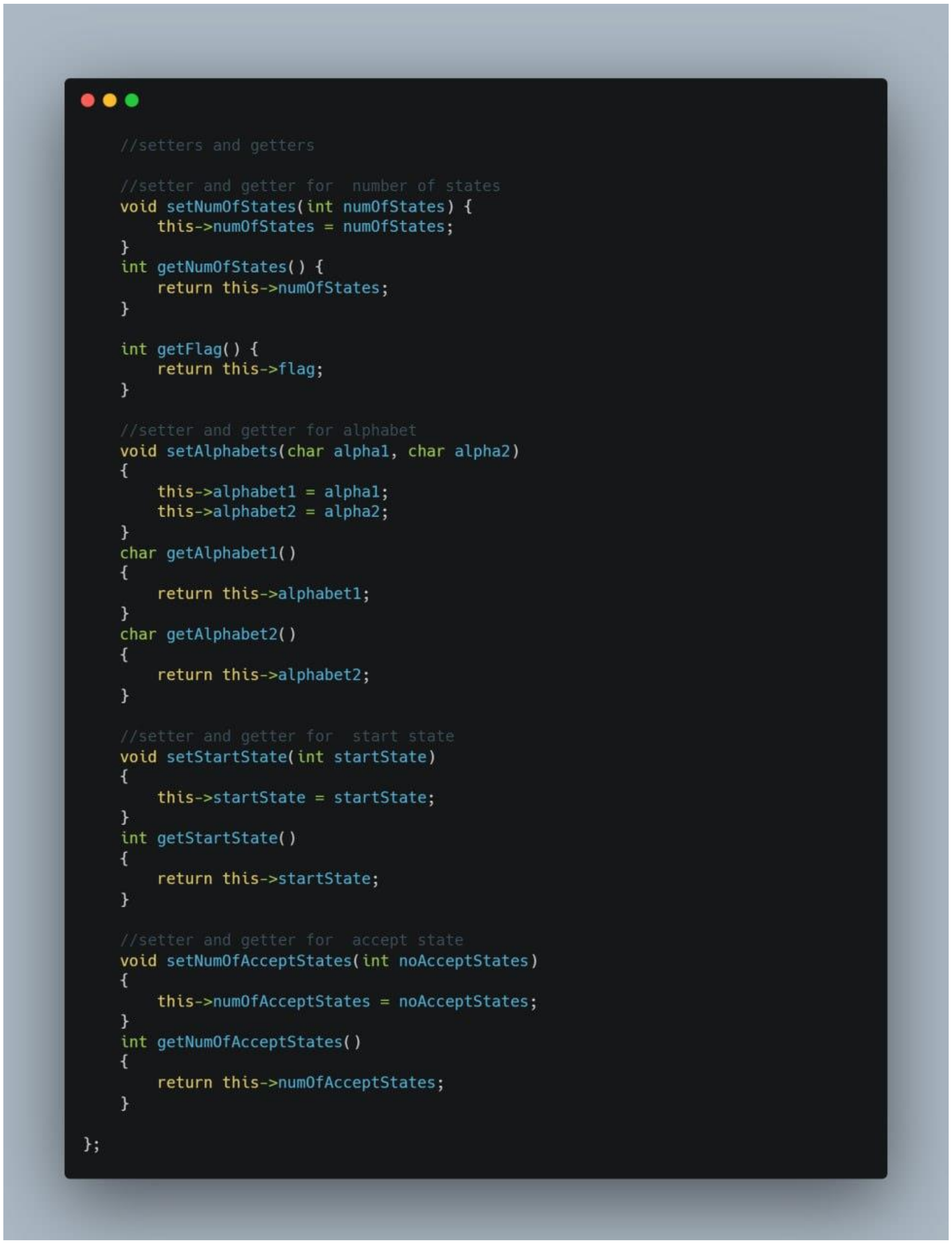
In Figure 3.2.2 we included the formal definition of the DFA machine.



```
class DFA {

private:
    int numOfStates;
    vector<int>states;
    char alphabet1, alphabet2;
    int startState;
    int numOfAcceptStates;
    vector<int>acceptStates;
    map<pair<int, char>, int>transitionsOfStates;
    int flag = 0;
```

Figure 3.2.2

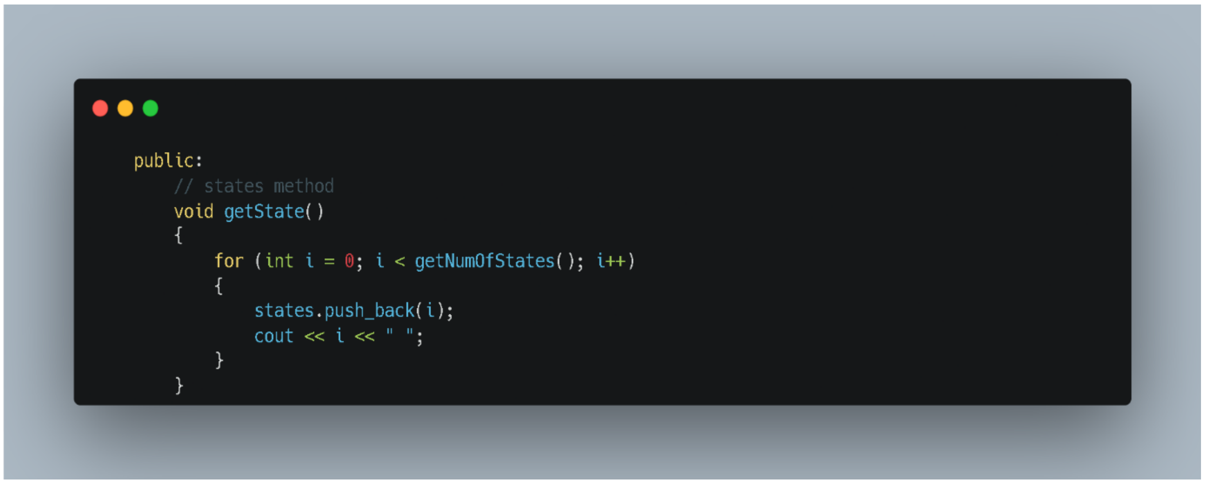
In Figure 3.2.3 there are setters and getters methods for number of states, alphabets, start state, and accept states



```
    //setters and getters

    //setter and getter for  number of states
    void setNumOfStates(int numOfStates) {
        this->numOfStates = numOfStates;
    }
    int getNumOfStates() {
        return this->numOfStates;
    }

    int getFlag() {
        return this->flag;
    }

    //setter and getter for alphabet
    void setAlphabets(char alpha1, char alpha2)
    {
        this->alphabet1 = alpha1;
        this->alphabet2 = alpha2;
    }
    char getAlphabet1()
    {
        return this->alphabet1;
    }
    char getAlphabet2()
    {
        return this->alphabet2;
    }

    //setter and getter for  start state
    void setStartState(int startState)
    {
        this->startState = startState;
    }
    int getStartState()
    {
        return this->startState;
    }

    //setter and getter for  accept state
    void setNumOfAcceptStates(int noAcceptStates)
    {
        this->numOfAcceptStates = noAcceptStates;
    }
    int getNumOfAcceptStates()
    {
        return this->numOfAcceptStates;
    }

};
```

Figure 3.2.3

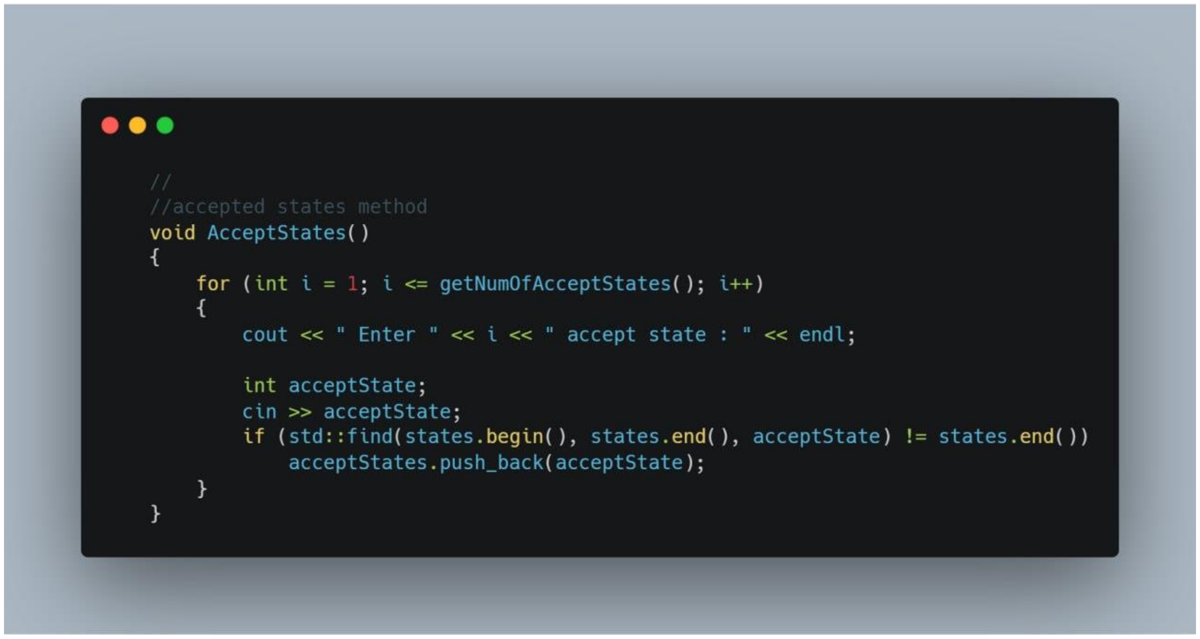
In Figure 3.2.4 the get states method initializes the set of states for the DFA and displays them in the console.



```
public:
    // states method
    void getState()
    {
        for (int i = 0; i < getNumOfStates(); i++)
        {
            states.push_back(i);
            cout << i << " ";
        }
    }
```

Figure 3.2.4

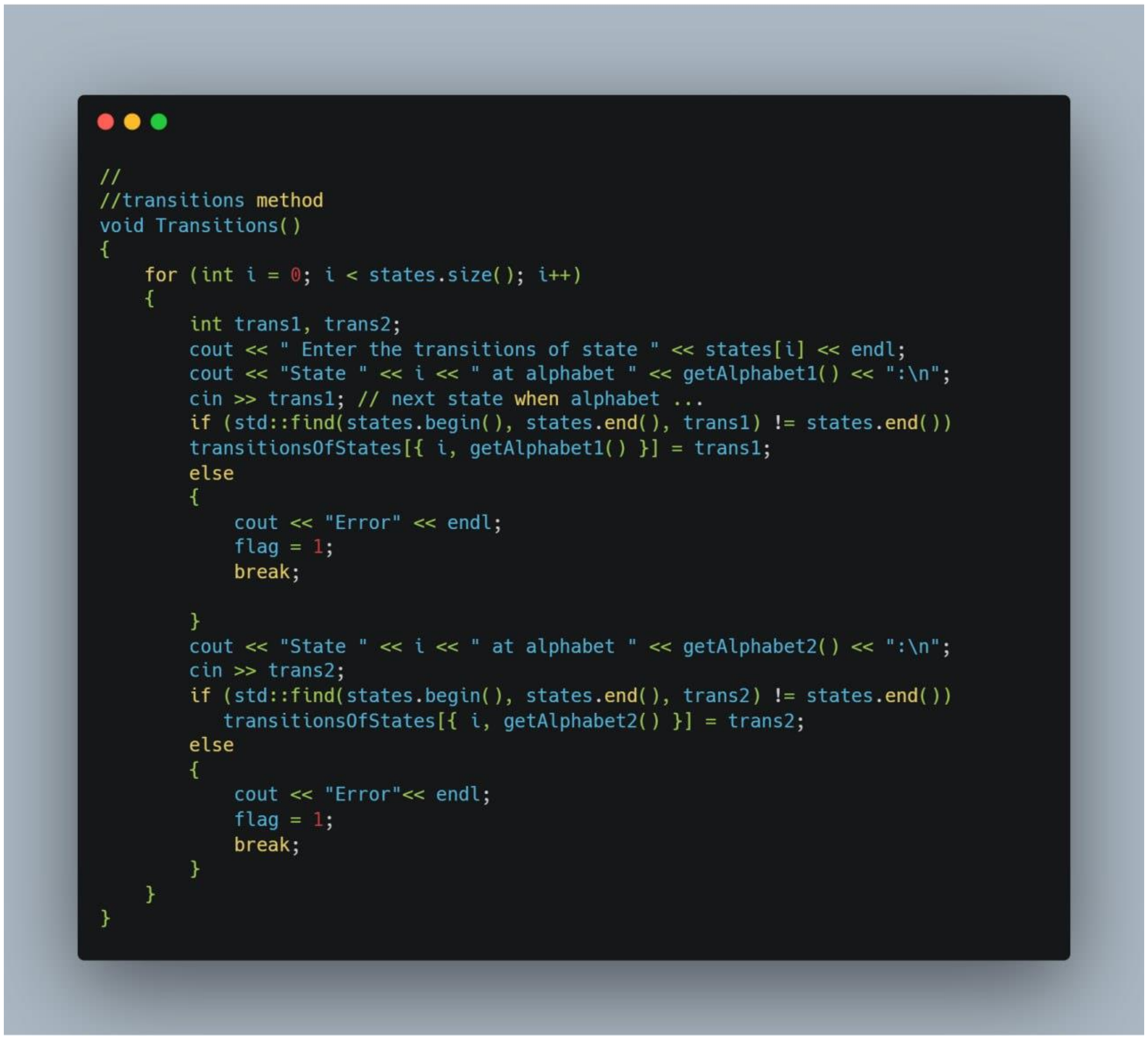
In Figure 3.2.5 the Accept States Method collects and stores the accept states for the DFA by verifying that the entered states are valid states within the DFA.



```
//
//accepted states method
void AcceptStates()
{
    for (int i = 1; i <= getNumOfAcceptStates(); i++)
    {
        cout << " Enter " << i << " accept state : " << endl;

        int acceptState;
        cin >> acceptState;
        if (std::find(states.begin(), states.end(), acceptState) != states.end())
            acceptStates.push_back(acceptState);
    }
}
```

Figure3.2.5

In Figure 3.2.6 the Transitions Method which is responsible for obtaining and validating the transition of each state and alphabets in the DFA.s

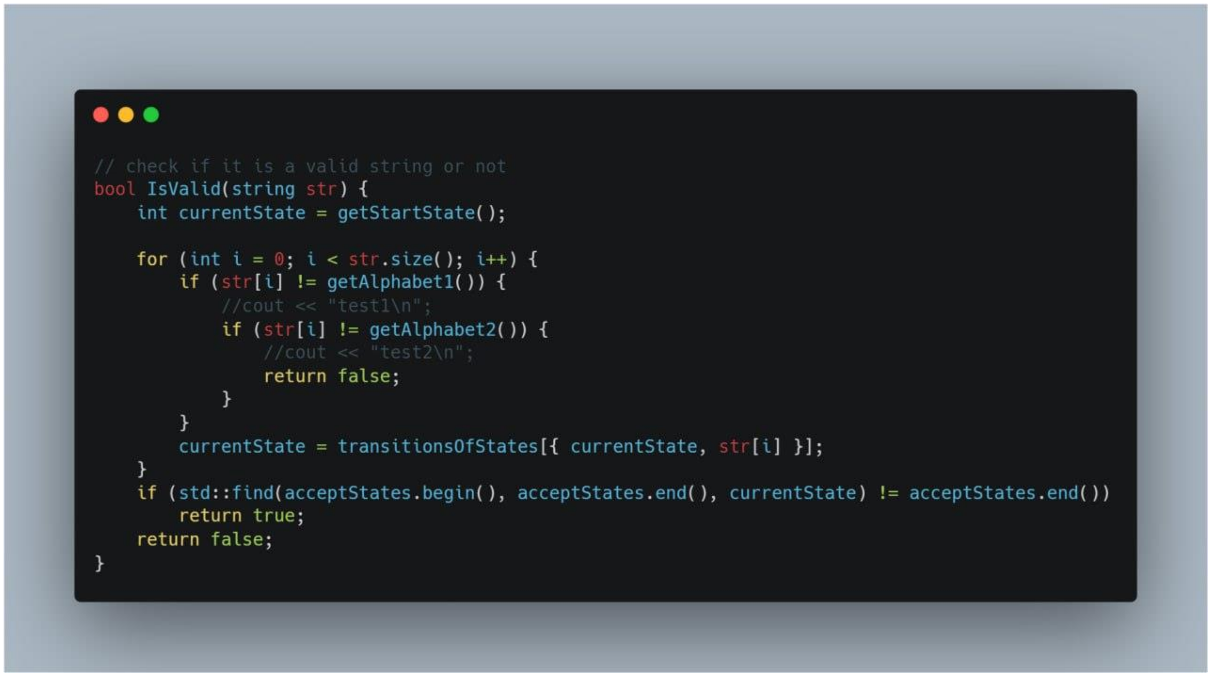


```
//
//transitions method
void Transitions()
{
    for (int i = 0; i < states.size(); i++)
    {
        int trans1, trans2;
        cout << " Enter the transitions of state " << states[i] << endl;
        cout << "State " << i << " at alphabet " << getAlphabet1() << ":\n";
        cin >> trans1; // next state when alphabet ...
        if (std::find(states.begin(), states.end(), trans1) != states.end())
        transitionsOfStates[{ i, getAlphabet1() }] = trans1;
        else
        {
            cout << "Error" << endl;
            flag = 1;
            break;

        }
        cout << "State " << i << " at alphabet " << getAlphabet2() << ":\n";
        cin >> trans2;
        if (std::find(states.begin(), states.end(), trans2) != states.end())
           transitionsOfStates[{ i, getAlphabet2() }] = trans2;
        else
        {
            cout << "Error"<< endl;
            flag = 1;
            break;
        }
    }
}
```

Figure 3.2.6

.

In Figure 3.2.7 the validation method checks whether a given string is accepted by the DFA according to its alphabets, transitions and accept states.



```
// check if it is a valid string or not
bool IsValid(string str) {
    int currentState = getStartState();

    for (int i = 0; i < str.size(); i++) {
        if (str[i] != getAlphabet1()) {
            //cout << "test1\n";
            if (str[i] != getAlphabet2()) {
                //cout << "test2\n";
                return false;
            }
        }
        currentState = transitionsOfStates[{ currentState, str[i] }];
    }
    if (std::find(acceptStates.begin(), acceptStates.end(), currentState) != acceptStates.end())
        return true;
    return false;
}
```

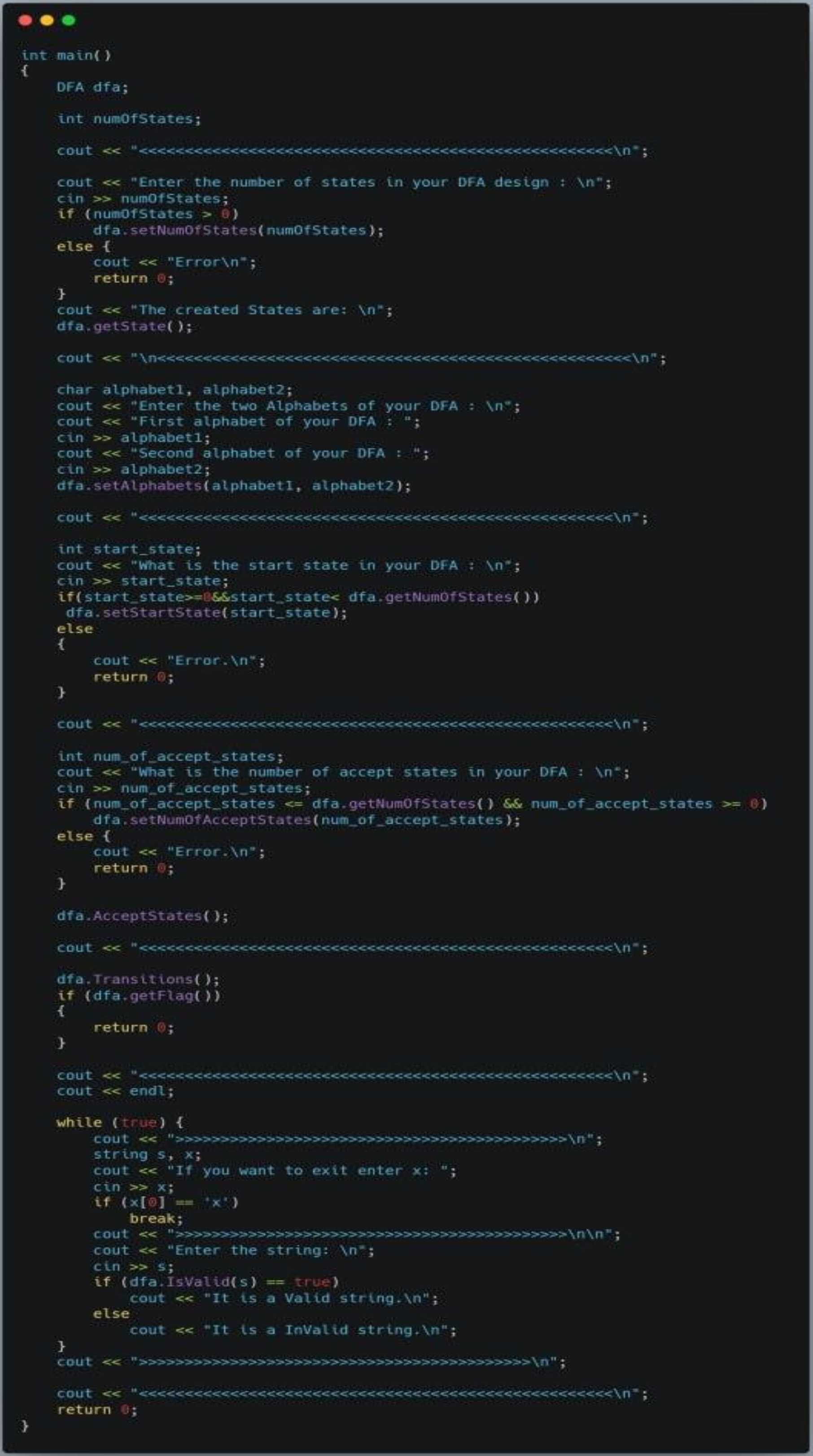
Figure 3.2.7

In Figure 3.2.8 the main which is the entry point of the program.

First, creating an instance of the DFA class, prompting the user to input information about the DFA, including the number of states, alphabet, start state number of accept states, accept states , and transitions of each state.

Second, call the relevant methods to set up the DFA.

Finally, entering an infinite loop allows the user to input strings for validation until the user decides to exit



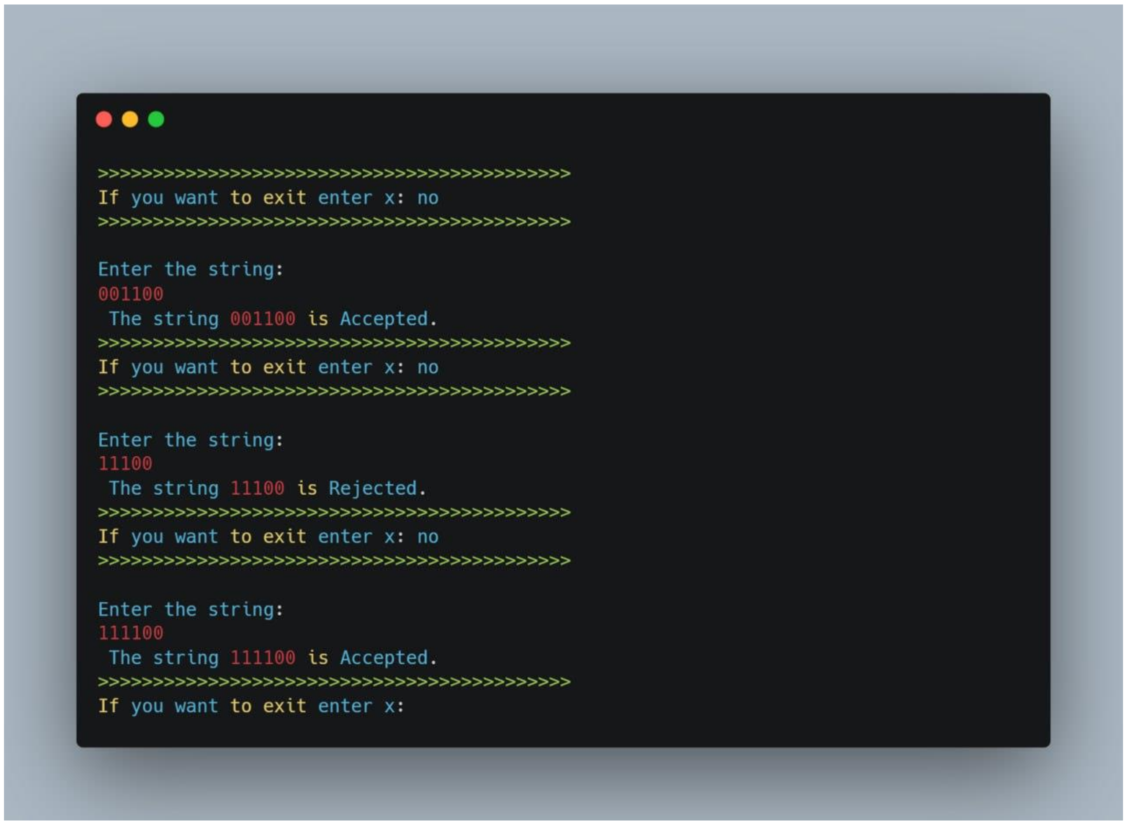
```cpp
int main()
{
    DFA dfa;

    int numOfStates;

    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";

    cout << "Enter the number of states in your DFA design : \n";
    cin >> numOfStates;
    if (numOfStates > 0)
        dfa.setNumOfStates(numOfStates);
    else {
        cout << "Error\n";
        return 0;
    }
    cout << "The created States are: \n";
    dfa.getState();

    cout << "\n<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";

    char alphabet1, alphabet2;
    cout << "Enter the two Alphabets of your DFA : \n";
    cout << "First alphabet of your DFA : ";
    cin >> alphabet1;
    cout << "Second alphabet of your DFA : ";
    cin >> alphabet2;
    dfa.setAlphabets(alphabet1, alphabet2);

    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";

    int start_state;
    cout << "What is the start state in your DFA : \n";
    cin >> start_state;
    if(start_state>=0&&start_state< dfa.getNumOfStates())
     dfa.setStartState(start_state);
    else
    {
        cout << "Error.\n";
        return 0;
    }

    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";

    int num_of_accept_states;
    cout << "What is the number of accept states in your DFA : \n";
    cin >> num_of_accept_states;
    if (num_of_accept_states <= dfa.getNumOfStates() && num_of_accept_states >= 0)
        dfa.setNumOfAcceptStates(num_of_accept_states);
    else {
        cout << "Error.\n";
        return 0;
    }

    dfa.AcceptStates();

    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";

    dfa.Transitions();
    if (dfa.getFlag())
    {
        return 0;
    }

    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";
    cout << endl;

    while (true) {
        cout << ">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n";
        string s, x;
        cout << "If you want to exit enter x: ";
        cin >> x;
        if (x[0] == 'x')
            break;
        cout << ">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n\n";
        cout << "Enter the string: \n";
        cin >> s;
        if (dfa.IsValid(s) == true)
            cout << "It is a Valid string.\n";
        else
            cout << "It is a InValid string.\n";
    }
    cout << ">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n";

    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";
    return 0;
}
```

Figure 3.2.8

3.3 Machine Examples:

3.3.1 Machine (1):

A machine that accepts any string that has even 0's and even 1's.

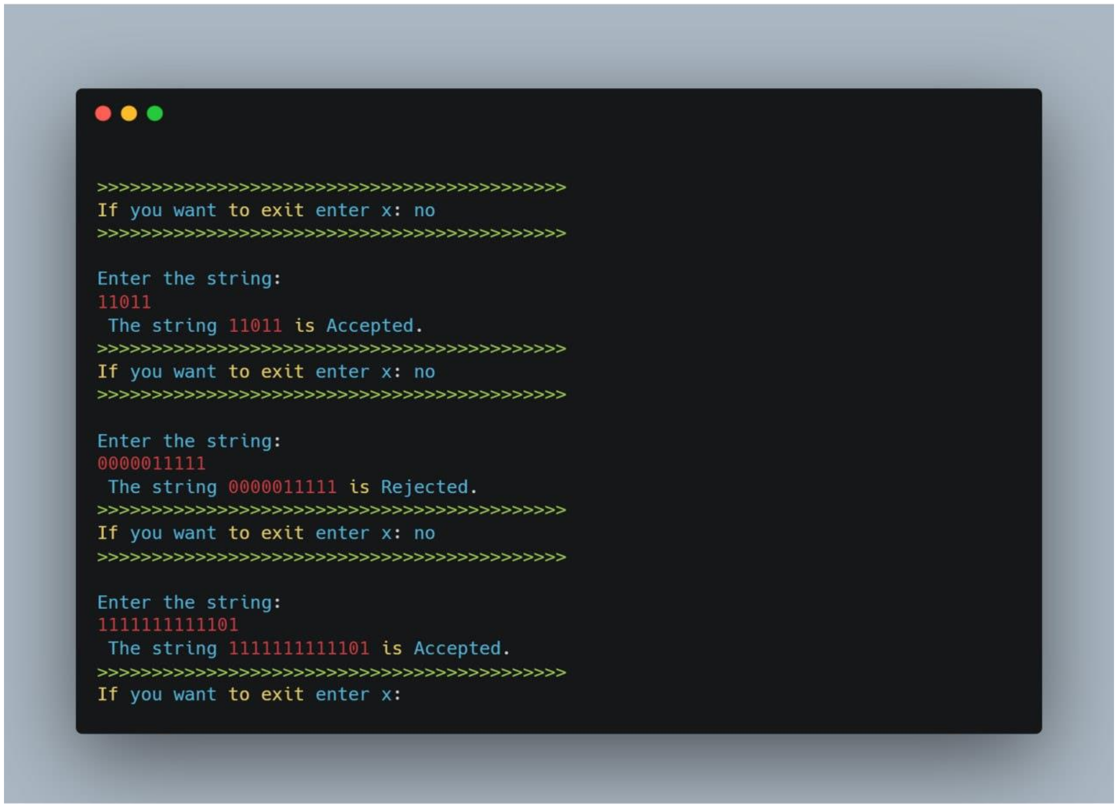


```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
If you want to exit enter x: no
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Enter the string:
001100
 The string 001100 is Accepted.
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
If you want to exit enter x: no
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Enter the string:
11100
 The string 11100 is Rejected.
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
If you want to exit enter x: no
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Enter the string:
111100
 The string 111100 is Accepted.
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
If you want to exit enter x:
```

Figure 3.3.1 accept and reject examples of machine 1

3.3.2 Machine (2):

A machine that accepts all the string over alphabets ∑ {0, 1} where each string contains “101” as a substring.

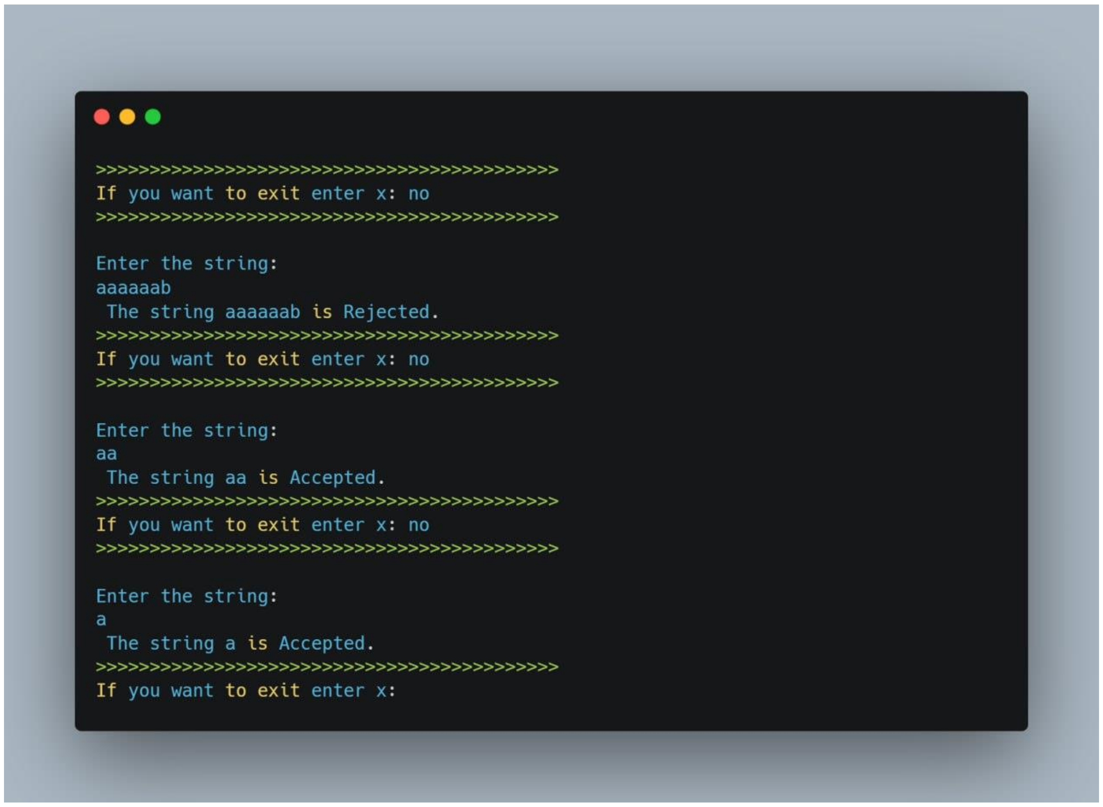


```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
If you want to exit enter x: no
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Enter the string:
11011
 The string 11011 is Accepted.
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
If you want to exit enter x: no
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Enter the string:
0000011111
 The string 0000011111 is Rejected.
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
If you want to exit enter x: no
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Enter the string:
1111111111101
 The string 1111111111101 is Accepted.
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
If you want to exit enter x:
```

Figure 3.3.2 accept and reject examples of machine 2.

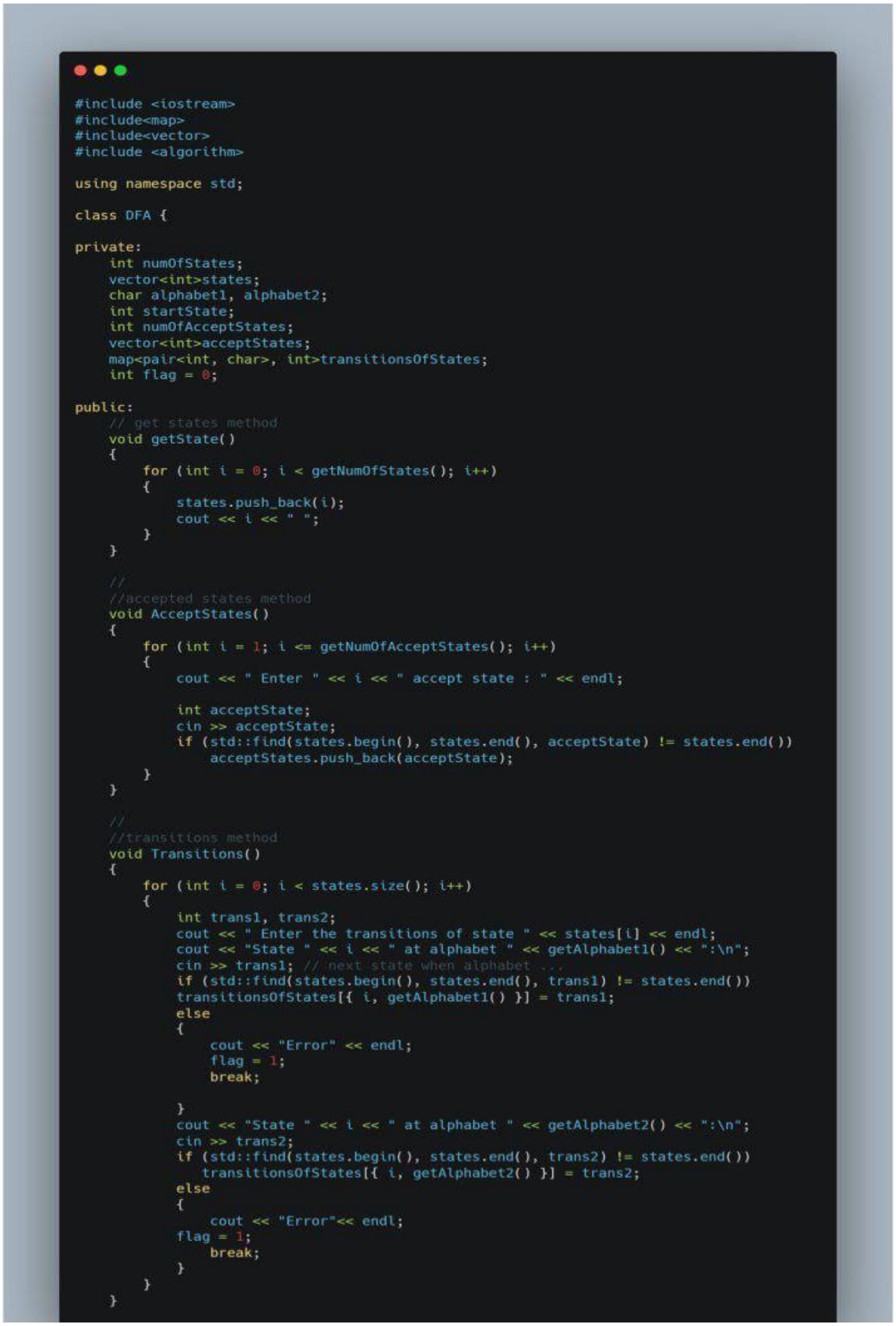
3.3.2 Machine (3):

A machine that accepts all the string that **starts and ends with the same symbol.**

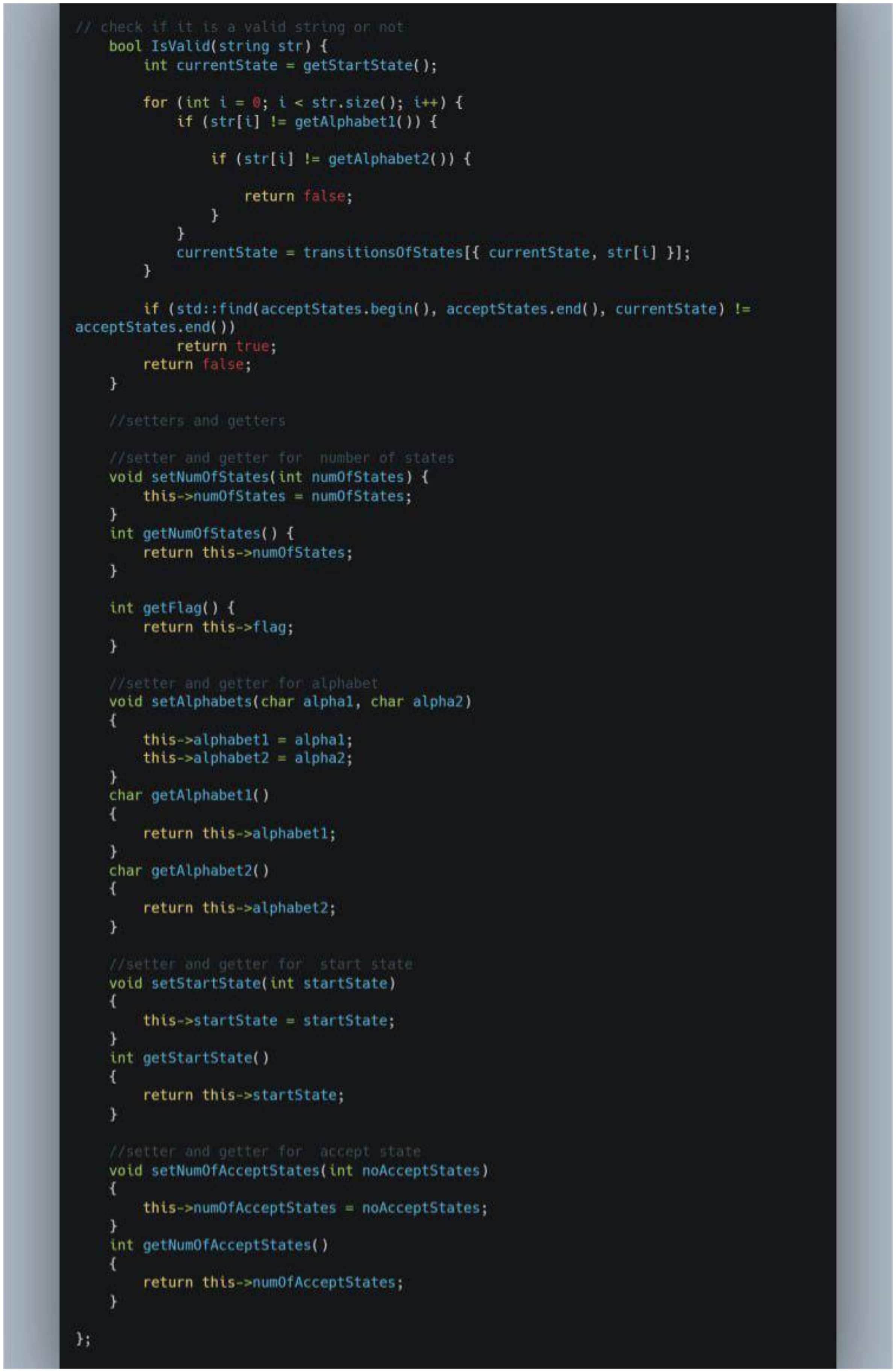


```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
If you want to exit enter x: no
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Enter the string:
aaaaaab
 The string aaaaaab is Rejected.
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
If you want to exit enter x: no
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Enter the string:
aa
 The string aa is Accepted.
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
If you want to exit enter x: no
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Enter the string:
a
 The string a is Accepted.
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
If you want to exit enter x:
```

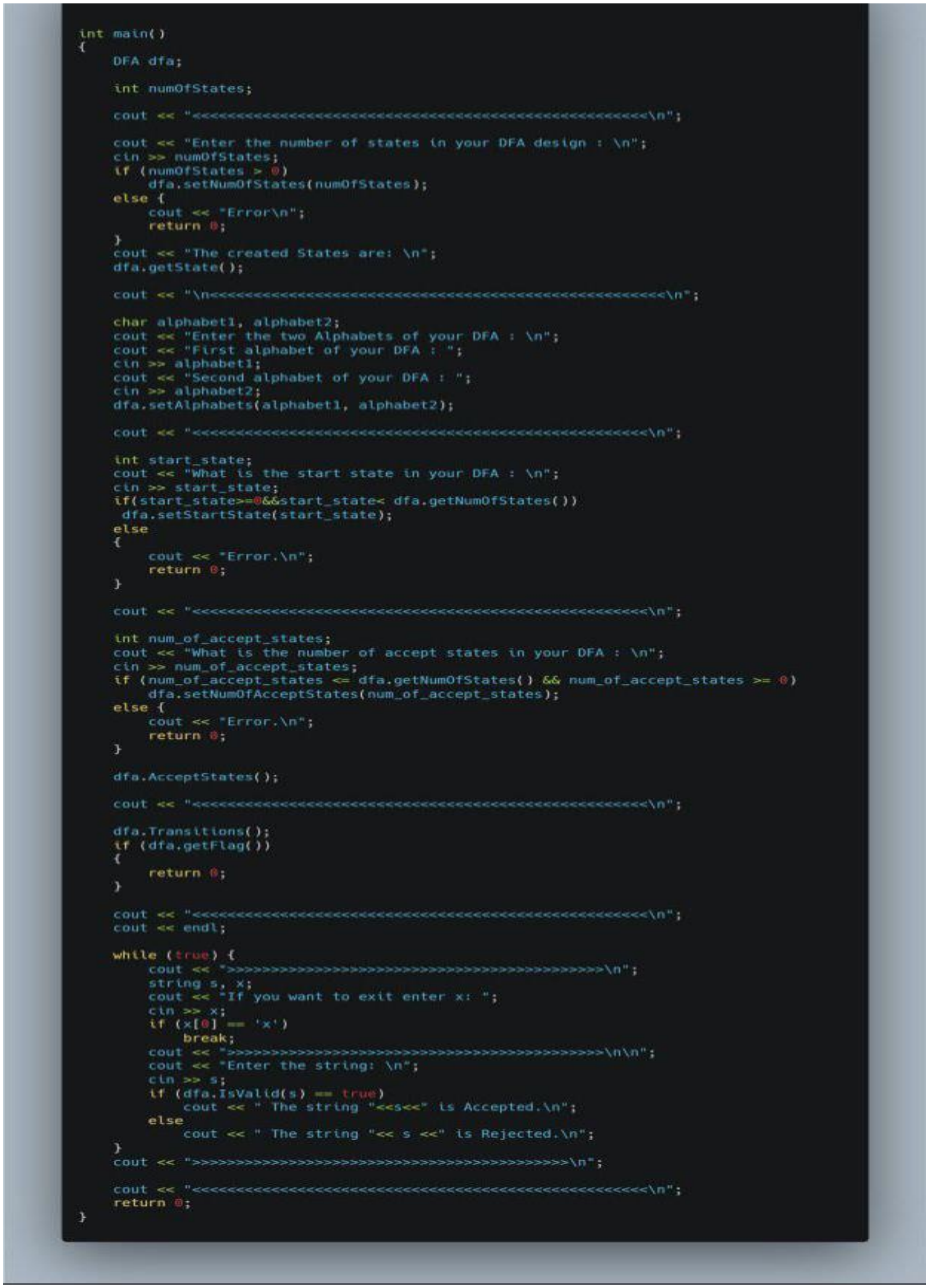
4. Appendix



```
#include <iostream>
#include<map>
#include<vector>
#include <algorithm>

using namespace std;

class DFA {

private:
    int numOfStates;
    vector<int>states;
    char alphabet1, alphabet2;
    int startState;
    int numOfAcceptStates;
    vector<int>acceptStates;
    map<pair<int, char>, int>transitionsOfStates;
    int flag = 0;

public:
    // get states method
    void getState()
    {
        for (int i = 0; i < getNumOfStates(); i++)
        {
            states.push_back(i);
            cout << i << " ";
        }
    }

    //
    //accepted states method
    void AcceptStates()
    {
        for (int i = 1; i <= getNumOfAcceptStates(); i++)
        {
            cout << " Enter " << i << " accept state : " << endl;

            int acceptState;
            cin >> acceptState;
            if (std::find(states.begin(), states.end(), acceptState) != states.end())
                acceptStates.push_back(acceptState);
        }
    }

    //
    //transitions method
    void Transitions()
    {
        for (int i = 0; i < states.size(); i++)
        {
            int trans1, trans2;
            cout << " Enter the transitions of state " << states[i] << endl;
            cout << "State " << i << " at alphabet " << getAlphabet1() << ":\n";
            cin >> trans1; // next state when alphabet ...
            if (std::find(states.begin(), states.end(), trans1) != states.end())
            transitionsOfStates[{ i, getAlphabet1() }] = trans1;
            else
            {
                cout << "Error" << endl;
                flag = 1;
                break;

            }
            cout << "State " << i << " at alphabet " << getAlphabet2() << ":\n";
            cin >> trans2;
            if (std::find(states.begin(), states.end(), trans2) != states.end())
               transitionsOfStates[{ i, getAlphabet2() }] = trans2;
            else
            {
                cout << "Error"<< endl;
            flag = 1;
                break;
            }
        }
    }
```



```
// check if it is a valid string or not
    bool IsValid(string str) {
        int currentState = getStartState();

        for (int i = 0; i < str.size(); i++) {
            if (str[i] != getAlphabet1()) {

                if (str[i] != getAlphabet2()) {

                    return false;
                }
            }
            currentState = transitionsOfStates[{ currentState, str[i] }];
        }

        if (std::find(acceptStates.begin(), acceptStates.end(), currentState) !=
acceptStates.end())
            return true;
        return false;
    }

    //setters and getters

    //setter and getter for  number of states
    void setNumOfStates(int numOfStates) {
        this->numOfStates = numOfStates;
    }
    int getNumOfStates() {
        return this->numOfStates;
    }

    int getFlag() {
        return this->flag;
    }

    //setter and getter for alphabet
    void setAlphabets(char alpha1, char alpha2)
    {
        this->alphabet1 = alpha1;
        this->alphabet2 = alpha2;
    }
    char getAlphabet1()
    {
        return this->alphabet1;
    }
    char getAlphabet2()
    {
        return this->alphabet2;
    }

    //setter and getter for  start state
    void setStartState(int startState)
    {
        this->startState = startState;
    }
    int getStartState()
    {
        return this->startState;
    }

    //setter and getter for  accept state
    void setNumOfAcceptStates(int noAcceptStates)
    {
        this->numOfAcceptStates = noAcceptStates;
    }
    int getNumOfAcceptStates()
    {
        return this->numOfAcceptStates;
    }

};
```



```
int main()
{
    DFA dfa;

    int numOfStates;

    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";

    cout << "Enter the number of states in your DFA design : \n";
    cin >> numOfStates;
    if (numOfStates > 0)
        dfa.setNumOfStates(numOfStates);
    else {
        cout << "Error\n";
        return 0;
    }
    cout << "The created States are: \n";
    dfa.getState();

    cout << "\n<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";

    char alphabet1, alphabet2;
    cout << "Enter the two Alphabets of your DFA : \n";
    cout << "First alphabet of your DFA : ";
    cin >> alphabet1;
    cout << "Second alphabet of your DFA : ";
    cin >> alphabet2;
    dfa.setAlphabets(alphabet1, alphabet2);

    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";

    int start_state;
    cout << "What is the start state in your DFA : \n";
    cin >> start_state;
    if(start_state>=0&&start_state< dfa.getNumOfStates())
     dfa.setStartState(start_state);
    else
    {
        cout << "Error.\n";
        return 0;
    }

    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";

    int num_of_accept_states;
    cout << "What is the number of accept states in your DFA : \n";
    cin >> num_of_accept_states;
    if (num_of_accept_states <= dfa.getNumOfStates() && num_of_accept_states >= 0)
        dfa.setNumOfAcceptStates(num_of_accept_states);
    else {
        cout << "Error.\n";
        return 0;
    }

    dfa.AcceptStates();

    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";

    dfa.Transitions();
    if (dfa.getFlag())
    {
        return 0;
    }

    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";
    cout << endl;

    while (true) {
        cout << ">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n";
        string s, x;
        cout << "If you want to exit enter x: ";
        cin >> x;
        if (x[0] == 'x')
            break;
        cout << ">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n\n";
        cout << "Enter the string: \n";
        cin >> s;
        if (dfa.IsValid(s) == true)
            cout << " The string "<<s<<" is Accepted.\n";
        else
            cout << " The string "<< s <<" is Rejected.\n";
    }
    cout << ">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n";

    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n";
    return 0;
}
```

https://ideone.com/l9JLIC

5. Project Plan

Task	Team member
Collecting information about Automata Theory and DFA	Enas Ikram
Definition and Design Machine 1	Moreen Mohsen
Definition and Design Machine 2	Aya Abdellatif
Definition and Design Machine 3	Jessica Ayman
Code implementation	Moreen Mohsen
Add Validation to the Code	Aya Abdellatif
Reference	Jessica Ayman

6. References

[1] *wikipedia* , accessed 12 December 2023,<https://en.wikipedia.org/wiki/Automata_theory>

[2] *javatpoint* , accessed 12 December 2023 ,<https://www.javatpoint.com/theory-of-automata>

[3]*cs.stanford.edu ,* accessed 15 December 2023,<https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html>

[4] *javatpoint*, accessed 17 December 2023,< https://www.javatpoint.com/examples-of-deterministic-finite-automata>

[5] cstaleem, accessed 1 January 2024,<https://cstaleem.com/examples-of-dfa>

[6] greeksforgeeks, accessed 1 January 2024,<https://www.geeksforgeeks.org/program-to-build-a-dfa-to-accept-strings-that-start-and-end-with-same-character/>