# Code Document

**Washington State University – Tri Cities**

**CPT_S 322: Software Engineering Principles I**

**Project:**
Turing Machine Console Application

**Course Instructor:**
Dr. Niel B. Corrigan

**Student:**

Raul Martinez

**Date:**

April 28th, 2025

## Introduction

This document contains all of the source code for the Turing Machine Console Application, this is the final version of the code that will be officially tested and graded by Dr. Niel B. Corrigan. All cpp and header files are be included. The language for this application is C++.

**Changes: Tape.cpp, Turing_Machine.h, Turing_Machine.cpp**

## Crash.cpp:

```
//verified on 4/14

#include "crash.h"

#include <stdexcept>
#include <string>

using namespace std;

crash::crash(string reason):
    runtime_error(reason.c_str())
    {

    }
```

## Crash.h:

```
//verified on 4/14

#ifndef crash_h
#define crash_h

#include <stdexcept>
#include <string>

using namespace std;

class crash:public runtime_error

{
    public:
        crash(string reason);
};

#endif
```

## Direction.h:

```
//verified on 4/14
#ifndef direction_h
#define direction_h

typedef char direction;

#endif
```

## Final_state.cpp:

```
//final
#include "final_state.h"
#include "string_pointer.h"

int final_state::number_of_states = 0;

final_state::final_state()
{
    name = new std::string;
    ++number_of_states;
}

final_state::final_state(const std::string& state_name)
{
    name = new std::string(state_name);
    ++number_of_states;
}

final_state::final_state(const final_state& other)
{
    name = new std::string(*other.name);
    ++number_of_states;
}

final_state::~final_state()
{
    delete name;
    --number_of_states;
}

final_state& final_state::operator=(const final_state& other)
{
    if (this != &other)
    {
        *name = *other.name;
    }
    return *this;
}

void final_state::get_name(std::string& state_name) const
{
    state_name = *name;
}

void final_state::set_name(const std::string& state_name)
{
    *name = state_name;
}

int final_state::total_number_of_states()
{
    return number_of_states;
}
```

## Final_state.h:

```
//final
#ifndef final_state_h
#define final_state_h

#include "string_pointer.h"
#include <string>

class final_state
{
private:
    string_pointer name;
    static int number_of_states;

public:
    final_state();
    final_state(const std::string& state_name);
    final_state(const final_state& other);
    virtual ~final_state();

    final_state& operator=(const final_state& other);

    void get_name(std::string& state_name) const;
    void set_name(const std::string& state_name);
    static int total_number_of_states();
};

typedef final_state* final_state_pointer;

#endif
```

## Final_states.cpp:

```
//final?
#include "final_states.h"
#include "final_state.h"
#include "states.h"
#include "crash.h"
#include "string_play.h"
#include "uppercase.h"

#include <vector>
#include <fstream>
#include <string>
#include <sstream>
#include <iostream>

void final_states::load(std::ifstream& definition, bool& valid)
{
    std::string line;
    std::stringstream parseme;
    std::string current;

    // Search for the line starting with FINAL_STATES:
    while (std::getline(definition, line))
    {
        if (trim(line, " \t").empty())
            continue;

        parseme.clear();
        parseme.str(line);
        parseme >> current;

        if (to_uppercase_string(current) == "FINAL_STATES:")
            break;
    }

    if (to_uppercase_string(current) != "FINAL_STATES:")
    {
        std::cout << "Final States Not Defined." << std::endl;
        valid = false;
        return;
    }

    // Read the remaining tokens in the line as final states
    while (parseme >> current)
    {
        final_state st(current);
        state_list.push_back(st);
    }
}

void final_states::validate(const states& all_states, bool& valid) const
{
    for (size_t i = 0; i < state_list.size(); ++i)
    {
        std::string name;
        state_list[i].get_name(name);

        if (!all_states.is_element(name))
        {
            std::cout << "Error: Final State '" << name << "' is not declared in the
list of states." << std::endl;
            valid = false;
        }

        for (size_t j = i + 1; j < state_list.size(); ++j)
        {
            std::string other_name;
            state_list[j].get_name(other_name);
            if (name == other_name)
            {
                std::cout << "Warning: duplicate final state '" << name << "' found."
<< std::endl;
                valid = false;
            }
        }
    }
}

void final_states::view() const
{
    std::cout << "F = { ";
    if (!state_list.empty())
    {
        std::string name;
        state_list[0].get_name(name);
        std::cout << name;
        for (size_t i = 1; i < state_list.size(); ++i)
        {
            state_list[i].get_name(name);
            std::cout << ", " << name;
        }
    }
    std::cout << " }" << std::endl << std::endl;
}

bool final_states::is_element(const std::string& state) const
{
    for (size_t i = 0; i < state_list.size(); ++i)
    {
        std::string name;
        state_list[i].get_name(name);
        if (name == state)
            return true;
    }
    return false;
}
```

### Final_states.h:

```
//final
#ifndef final_states_h
#define final_states_h

#include "final_state.h"
#include "states.h"

#include <vector>
#include <fstream>
#include <string>

class final_states
{
public:
    void load(std::ifstream& definition, bool& valid);
    void validate(const states& all_states, bool& valid) const;
    void view() const;
    bool is_element(const std::string& state) const;

private:
    std::vector<final_state> state_list;
};

#endif
```

## Input_alphabet.cpp:

```
//final?
#include "input_alphabet.h"
#include "string_play.h"
#include "uppercase.h"

#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>

void input_alphabet::load(std::ifstream& definition, bool& valid)
{
    std::string line;
    std::stringstream parseme;
    std::string current;

    // Search for INPUT_ALPHABET: line
    while (std::getline(definition, line))
    {
        if (trim(line, " \t").empty())
            continue;

        parseme.clear();
        parseme.str(line);
        parseme >> current;

        if (to_uppercase_string(current) == "INPUT_ALPHABET:")
            break;
    }

    if (to_uppercase_string(current) != "INPUT_ALPHABET:")
    {
        std::cout << "Input Alphabet Not Defined." << std::endl;
        valid = false;
        return;
    }

    // Read rest of the line for characters
    while (parseme >> current)
    {
        if (current.length() != 1)
        {
            std::cout << "Invalid Symbol in Input Alphabet: \"" << current << "\"" <<
std::endl;
```

```
            valid = false;
        }
        else if (!is_element(current[0]))
        {
            alphabet.push_back(current[0]);
        }
    }
}

void input_alphabet::view() const
{
    std::cout << "\u03A3 = { ";
    if (!alphabet.empty())
    {
        std::cout << alphabet[0];
        for (size_t i = 1; i < alphabet.size(); ++i)
        {
            std::cout << ", " << alphabet[i];
        }
    }
    std::cout << " }" << std::endl << std::endl;
}

bool input_alphabet::is_element(const char& value) const
{
    for (char ch : alphabet)
    {
        if (ch == value)
            return true;
    }
    return false;
}
```

## Input_alphabet.h:

```
//final
#ifndef input_alphabet_h
#define input_alphabet_h

#include <iostream>
#include <fstream>
#include <string>
#include <vector>

class input_alphabet
{
private:
    std::vector<char> alphabet;
public:
    void load(std::ifstream& definition, bool& valid);
    void view() const;
    bool is_element(const char& value) const;
};

#endif
```

## Main.cpp:

```
//final?
#include "turing_machine.h"
#include "visible.h"

#include <string>
#include <iostream>
#include <fstream>
#include <chrono>
#include <thread>
#include <cctype>

using namespace std;

// --- Utility Functions ---
//C++
char get_key_input()
{
    char str[1024];
    cin.getline(str, 1024);
    if (str[0] == '\0') return '-';
    if (str[1] != '\0') return '\0';
    return str[0];
}

void show_help()
{
    cout << endl;
    cout << "----------------------------------------------------------------" << endl;
    cout << "Input Chars |Commands     |Description                          " << endl;
    cout << "----------------------------------------------------------------" << endl;
    cout << "    (D)     |Delete       |Delete input string from list        " << endl;
    cout << "    (X)     |Exit         |Exit application                     " << endl;
    cout << "    (H)     |Help         |Help user                            " << endl;
    cout << "    (I)     |Insert       |Insert input string into list        " << endl;
    cout << "    (L)     |List         |List input strings                   " << endl;
    cout << "    (Q)     |Quit         |Quit operation of Turing machine     " << endl;
    cout << "    (R)     |Run         |Run Turing machine on input string   " << endl;
    cout << "    (E)     |Set         |Set number of transitions to perform " << endl;
    cout << "    (W)     |Show        |Show status of application           " << endl;
    cout << "    (T)     |Truncate    |Truncate instantaneous descriptions  " << endl;
    cout << "    (V)     |View        |View Turing machine definition       " << endl;
    cout << "---------------------------------------------------------------" << endl;
    cout << endl;
}

void show_banner()
{
    cout << endl;
    cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" << endl;
    cout << "  ********  CPT_S 322 Turing Machine Simulator  *********   " << endl;
    cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" << endl;
    cout << "         Created by: Raul Y. Martinez, 4/8/2025             " << endl;
    cout << "    for CPT_S 322 S2025 w/ Dr. Niel B. Corrigan @ WSU-TC    " << endl;
    cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" << endl;
    cout << endl;
}

void show_loading_animation(const string& message = "Loading", int dots = 4)
{
    cout << message;
    cout.flush();
    for (int i = 0; i < dots; ++i)
    {
        this_thread::sleep_for(chrono::milliseconds(500));
        cout << "." << flush;
    }
    cout << endl << endl;
}

bool read_index(const string& prompt, int& index)
{
    cout << prompt;
    cin >> index;
    if (cin.fail()) {
        cin.clear();
        cin.ignore(10000, '\n');
        cout << "Invalid Input, Please Enter an Integer." << endl;
        return false;
    }
    cin.ignore();
    return true;
}

// --- Main Program ---
```

```
int main(int argc, char* argv[])
{
    const int success = 0;

    if (argc != 2)
    {
        cout << "Usage: " << argv[0] << " tm" << endl;
        cout << "NOTE: The Proper Way to Start This Program is With './tm anbn',
(anbn) Will Automatically Link Both .def and .str Files of Name anbn.def/.str." <<
endl;
        return success;
    }

    try
    {
        int transitions_per_run = 1;
        bool help_enabled = true;
        int max_characters = 32;

        vector<string> list_of_input_strings;
        string def_file = string(argv[1]) + ".def";
        string str_file = string(argv[1]) + ".str";

        turing_machine tm(def_file);
        if (!tm.is_valid_definition()) return success;

        cout << endl;
        show_loading_animation();
        cout << endl;
        show_banner();
        show_help();
        cout << endl;
        cout << "Turing Machine Loaded Successfully! " << endl;
        cout << endl;

        ifstream input_stream(str_file);
        if (input_stream.good())
        {
            string line;
            int line_num = 1;
            while (getline(input_stream, line))
            {
                if (tm.is_valid_input_string(line))
                    list_of_input_strings.push_back(line);
                else
                    cout << "Input File: Ignored Bad Line: # " << line_num << endl;
                ++line_num;
            }
            input_stream.close();
        }

        char command;
        bool changes_made = false;

        do
        {
            cout << "COMMAND: ";
            command = tolower(get_key_input());

            int index = 0;
            int new_value = 0;
            bool init_success = true;
            string input;

            switch (command)
            {
                case 'd':
                    if (help_enabled)
                        cout << "Enter an Index (1 - " << list_of_input_strings.size()
<< ") to Delete:" << endl;
                    if (read_index("Delete Index: ", index) &&
                        index >= 1 && index <= list_of_input_strings.size())
                    {
                        list_of_input_strings.erase(list_of_input_strings.begin() +
index - 1);
                        changes_made = true;
                    }
                    else cout << "Invalid Index." << endl;
                    break;

                case 'x':
                    break;

                case 'h':
                    help_enabled = !help_enabled;
                    cout << "Help " << (help_enabled ? "Enabled" : "Disabled") << "."
<< endl;
                    if (help_enabled) show_help();
                    break;

                case 'i':
                    if (help_enabled)
                        cout << "Enter a String With Valid Input Alphabet Characters:"
<< endl;
                    cout << "Enter String: ";
                    getline(cin, input);
                    if (tm.is_valid_input_string(input))
                    {
                        list_of_input_strings.push_back(input);
                        changes_made = true;
                        cout << "String Inserted at Position " <<
list_of_input_strings.size() << endl;
                    }
                    else cout << "Invalid Input String." << endl;
                    break;

                case 'l':
                    if (help_enabled)
                        cout << "List of Input Strings:" << endl;
                    for (size_t i = 0; i < list_of_input_strings.size(); ++i)
                        cout << (i + 1) << ": " << visible(list_of_input_strings[i])
<< endl;
                    break;

                case 'q':
                    if (help_enabled)
                        cout << "Terminating Current Turing Machine" << endl;
                    tm.terminate_operation();
                    cout << "Quitting. ";
                    if (tm.is_accepted_input_string())
                        cout << "String Accepted";
                    else if (tm.is_rejected_input_string())
                        cout << "String Rejected";
                    else
                        cout << "String Not Accepted, OR Rejected ";
                    cout << " In " << tm.total_number_of_transitions() << "
Transitions." << endl;
                    break;

                case 'r':
                    if (help_enabled)
                        cout << "Running Turing Machine..." << endl;
                    if (!tm.is_used())
                    {
                        if (list_of_input_strings.empty())
                            cout << "No Input Strings Available." << endl;
                        else
                        {
                            if (help_enabled)
                                {
                                    cout << "Select a String Index to Run:" << endl;
                                    cout << endl;
                                    cout << "List of Input Strings:" << endl;
                                    for (size_t i = 0; i <
list_of_input_strings.size(); ++i)
                                        cout << (i + 1) << ": " <<
visible(list_of_input_strings[i]) << endl << endl;
                                }

                            if (read_index("Select Input String: ", index) &&
                                index >= 1 && index <= list_of_input_strings.size())
                            {
                                init_success =
tm.initialize(list_of_input_strings[index - 1]);
                            }
                            else
                            {
                                cout << "Invalid Index." << endl;
                                break;
                            }
                        }
                    }
                    if (tm.is_used() && init_success)
                    {
                        if (tm.total_number_of_transitions() == 0)
                        {
                            cout << "0. ";
                            tm.view_instantaneous_description(max_characters);
                        }
                        tm.perform_transitions(transitions_per_run);
                        cout << tm.total_number_of_transitions() << ". ";
                        tm.view_instantaneous_description(max_characters);
                        if (!tm.is_operating())
                        {
                            cout << (tm.is_accepted_input_string() ? "Accepted " :
"Rejected ")
                                 << "String \"" << tm.input_string() << "\" in "
                                 << tm.total_number_of_transitions() << "
Transitions." << endl;
                        }
                    }
                    break;

                case 'e':
                    if (help_enabled)
                        cout << "Set Transitions Per Run (Current: " <<
transitions_per_run << "):" << endl;
                    if (read_index("New Nalue: ", new_value) && new_value > 0)
                        transitions_per_run = new_value;
                    else cout << "Value MUST be a Positive Integer." << endl;
                    break;
                case 'w':
                    if (help_enabled) cout << endl;
                    cout << "------ Program Info ------" << endl;
                    cout << "CPT_S 322 - Spring 2025" << endl;
                    cout << "Dr. Corrigan @ WSU Tri-Cities" << endl;
                    cout << "Author: Raul Y. Martinez" << endl;
                    cout << "Help: " << (help_enabled ? "ON" : "OFF") << endl;
                    cout << "Max Characters Displayed: " << max_characters << endl;
                    cout << "Transitions Per Run: " << transitions_per_run << endl;
                    cout << "Version: 1.0 Beta" << endl;
                    cout << "Current String: \"" << tm.input_string() << "\" after "
<< tm.total_number_of_transitions() << " transitions." << endl;
                    break;

                case 't':
                    if (help_enabled)
                        cout << "Set Display Truncation (Current: " << max_characters
<< "):" << endl;
                    if (read_index("New Truncation: ", new_value) && new_value > 0)
                        max_characters = new_value;
                    else cout << "Must be a Positive Integer." << endl;
                    break;

                case 'v':
                    if (help_enabled)
                        cout << "Turing Machine Definition: " << endl;
                    tm.view_definition();
                    break;

                case '-':
                    break;

                default:
                    cout << "Unknown Command." << endl;
                    if (!help_enabled)
                        cout << "HINT: Press 'h' to Enable Help." << endl;
                    break;
            }

            cout << endl;
        }
        while (command != 'x');

        show_loading_animation("Exiting");

        if (changes_made)
        {
            ofstream out(str_file);
            for (const string& str : list_of_input_strings)
                out << str << endl;
            cout << "Saved Updated Input Strings To: " << str_file << endl;
        }
    }
    catch (exception& e)
    {
        cout << "Exception: " << e.what() << endl;
    }

    return success;
}
```

## Makefile:

```
# Compiler and options
CC = g++
OPTIONS = -g -Wall

# Main executable
all: tm

# Test executables
test: main_state_test main_tape_test

# Object file rules
string_play.o: string_play.cpp string_play.h
    $(CC) $(OPTIONS) -c $< -o $@

tape.o: tape.cpp tape.h
    $(CC) $(OPTIONS) -c $< -o $@

state.o: state.cpp state.h
    $(CC) $(OPTIONS) -c $< -o $@

final_state.o: final_state.cpp final_state.h
    $(CC) $(OPTIONS) -c $< -o $@

final_states.o: final_states.cpp final_states.h
    $(CC) $(OPTIONS) -c $< -o $@

crash.o: crash.cpp crash.h
    $(CC) $(OPTIONS) -c $< -o $@

transition.o: transition.cpp transition.h
    $(CC) $(OPTIONS) -c $< -o $@
```

```
transition_function.o: transition_function.cpp transition_function.h
    $(CC) $(OPTIONS) -c $< -o $@

visible.o: visible.cpp visible.h
    $(CC) $(OPTIONS) -c $< -o $@

turing_machine.o: turing_machine.cpp turing_machine.h
    $(CC) $(OPTIONS) -c $< -o $@

input_alphabet.o: input_alphabet.cpp input_alphabet.h
    $(CC) $(OPTIONS) -c $< -o $@

tape_alphabet.o: tape_alphabet.cpp tape_alphabet.h
    $(CC) $(OPTIONS) -c $< -o $@

states.o: states.cpp states.h
    $(CC) $(OPTIONS) -c $< -o $@

# Main program object
tm.o: main.cpp
    $(CC) $(OPTIONS) -c $< -o $@

# Final executable build
tm: tm.o tape.o crash.o state.o final_state.o final_states.o states.o \
    transition.o transition_function.o visible.o turing_machine.o \
    input_alphabet.o tape_alphabet.o string_play.o
    $(CC) $(OPTIONS) $^ -o $@

# State test
main_state_test.o: main_state_test.cpp
    $(CC) $(OPTIONS) -c $< -o $@

main_state_test: main_state_test.o state.o
    $(CC) $(OPTIONS) $^ -o $@

# Tape test
main_tape_test.o: main_tape_test.cpp
    $(CC) $(OPTIONS) -c $< -o $@

main_tape_test: main_tape_test.o tape.o crash.o
    $(CC) $(OPTIONS) $^ -o $@

# Clean target
clean:
    @rm -f tm main_state_test main_tape_test *.o
```

## State.cpp:

```
//verified 4/14
#include "state.h"
#include "string_pointer.h"

#include <string>

using namespace std;

int state::number_of_states = 0;

state::state()
{
    name = new string;
    ++number_of_states;
}

state::state(string state_name)
{
    name = new string(state_name);
    ++number_of_states;
}

state::state(const state& state)
{
    name = new string(*state.name);
    ++number_of_states;
}

state::~state()
{
    delete name;
    --number_of_states;
}

state & state::operator=(const state& state)
{
    if (this != &state)
    {
        *name = *state.name;
    }
    return *this;
}

void state::get_name(string& state_name) const
{
    state_name = *name;
}

void state::set_name(string state_name)
{
    *name = state_name;
}

int state::total_number_of_states()
{
    return number_of_states;
}
```

## State.h:

```
//verified 4/14
#ifndef state_h
#define state_h

#include "string_pointer.h"
#include <string>

using namespace std;

class state
{
    private:
        string_pointer name;
        static int number_of_states;

    public:
        state();
        state(string state_name);
        state(const state& state);
        virtual ~state();

        state& operator=(const state& state);

        void get_name(string& state_name) const;
        void set_name(string state_name);

        static int total_number_of_states();
};

typedef state *state_pointer;

#endif
```

### States.cpp:

```
//final?
#include "states.h"
#include "state.h"
#include "crash.h"
#include "uppercase.h"
#include "string_play.h"

#include <vector>
#include <fstream>
#include <string>
#include <sstream>
#include <iostream>

void states::load(std::ifstream& definition, bool& valid)
{
    std::string line;
    std::string token;

    // Look for the "STATES:" line
    while (std::getline(definition, line))
    {
        line = trim(line, " \t");
        if (line.empty()) continue;

        std::stringstream parseme(line);
        parseme >> token;

        if (uppercase_string(token) == "STATES:")
        {
            // Load states from the same line
            while (parseme >> token)
            {
                state st(token);
                state_list.push_back(st);
            }

            return; // Successfully loaded
        }
    }

    std::cout << "ERROR: States Not Defined " << std::endl;
    valid = false;
}

void states::view() const
{
    std::cout << "Q = { ";
    if (!state_list.empty())
    {
        std::cout << element(0);
        for (int i = 1; i < size(); ++i)
        {
            std::cout << ", " << element(i);
        }
    }
    std::cout << " }\n\n";
}

int states::size() const
{
    return static_cast<int>(state_list.size());
}

std::string states::element(int index) const
{
    if (index >= size())
        throw crash("States' Index Out-of-Bounds. ");

    std::string name;
    state_list[index].get_name(name);
    return name;
}

bool states::is_element(const std::string& state_name) const
{
    std::string target = uppercase_string(state_name);
    for (int i = 0; i < size(); ++i)
    {
        std::string name;
        state_list[i].get_name(name);
        if (uppercase_string(name) == target)
            return true;
    }
    return false;
}
```

## States.h:

```
//final?
#ifndef states_h
#define states_h

#include "state.h"
#include <vector>
```

```
#include <fstream>
#include <string>

class states
{
public:
    void load(std::ifstream& definition, bool& valid);
    void view() const;
    int size() const;
    std::string element(int index) const;
    bool is_element(const std::string& state) const;

private:
    std::vector<state> state_list;
};

#endif
```

## String_play.cpp:

```
//final?
#include <iostream>
#include <string>
#include <cctype>  // for toupper, tolower, isalnum, isspace

using namespace std;

// Returns a trimmed copy of the input string
string trim(const string& str, const string& whitespace = " \t")
{
    const size_t str_begin = str.find_first_not_of(whitespace);
    if (str_begin == string::npos)
        return "";  // no content

    const size_t str_end = str.find_last_not_of(whitespace);
    const size_t str_range = str_end - str_begin + 1;

    return str.substr(str_begin, str_range);
}

// Returns a copy of the string in uppercase
string to_uppercase_copy(const string& input)
{
    string result = input;
    for (char& c : result)
        c = toupper(c);
    return result;
}
```

```
// Returns a copy of the string in lowercase
string to_lowercase_copy(const string& input)
{
    string result = input;
    for (char& c : result)
        c = tolower(c);
    return result;
}

// Keeps only alphanumeric and whitespace characters in a string
string keep_alphanumeric_whitespace(const string& input)
{
    string result;
    for (char c : input)
    {
        if (isalnum(c) || isspace(c))
            result += c;
    }
    return result;
}
```

## String_play.h:

```
//final?
#ifndef string_play_h
#define string_play_h

#include <string>
#include <cctype>

// Trims whitespace from both ends of the input string
std::string trim(const std::string& str, const std::string& whitespace = " \t");

// Returns a new string converted to uppercase
std::string to_uppercase_copy(const std::string& input);

// Returns a new string converted to lowercase
std::string to_lowercase_copy(const std::string& input);

// Returns a new string with only alphanumeric and whitespace characters
std::string keep_alphanumeric_whitespace(const std::string& input);

#endif
```

## String_pointer.h:

```
//verified 4/14
#ifndef string_pointer_h
#define string_pointer_h

#include <string>
using namespace std;

typedef string *string_pointer;

#endif
```

## Tape_alphabet.cpp:

```
//final?
#include "tape_alphabet.h"
#include "string_play.h"
#include "uppercase.h"

#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>

void tape_alphabet::load(std::ifstream & definition, bool & valid)
{
    std::string line;
    std::stringstream parseme;
    std::string current;

    // Search for "TAPE_ALPHABET:"
    while (std::getline(definition, line))
    {
        if (trim(line, " \t").empty()) continue;

        parseme.clear();
        parseme.str(line);
        parseme >> current;

        if (uppercase_string(current) == "TAPE_ALPHABET:")
            break;
    }

    if (uppercase_string(current) != "TAPE_ALPHABET:")
    {
        std::cout << "ERROR: Tape Alphabet Not Defined or Misplaced. " << std:: endl;
        valid = false;
        return;
    }

    // Load symbols from the rest of the line
    while (parseme >> current)
    {
        if (!current.empty())
        {
            alphabet.push_back(current[0]);
        }
    }
}

void tape_alphabet::view() const
{
    std::cout << "\u0393 = { ";
    if (!alphabet.empty())
    {
        std::cout << alphabet[0];
        for (size_t i = 1; i < alphabet.size(); ++i)
        {
            std::cout << ", " << alphabet[i];
        }
    }
    std::cout << " }\n\n";
}

bool tape_alphabet::is_element(const char & value) const
{
    for (char ch : alphabet)
    {
        if (ch == value)
            return true;
    }
    return false;
}
```

## Tape_alphabet.h:

```
//final?
#ifndef tape_alphabet_h
#define tape_alphabet_h

#include <iostream>
#include <fstream>
#include <string>
```

```cpp
#include <vector>

using namespace std;

class tape_alphabet
{
private:
    vector<char> alphabet;

public:
    void load(ifstream & definition, bool & valid);
    void view() const;
    bool is_element(const char & value) const;
};

#endif
```

## Tape.cpp:

```cpp
//verified 4/14
#include "tape.h"
#include "input_alphabet.h"
#include "tape_alphabet.h"
#include "direction.h"
#include "uppercase.h"
#include "string_play.h"

#include <string>
#include <sstream>
#include <fstream>
#include <iostream>
#include <algorithm>

using namespace std;

tape::tape(): //verified 4/14
    cells(" "),
    current_cell(0),
    blank_character(' ')
{
}

void tape::load(ifstream& definition, bool& valid) // verified 4/14
{
    definition.clear();           // Clear any EOF or fail flags
    definition.seekg(0);          // Rewind to beginning

    string line;
    while (getline(definition, line))
    {
        if (trim(line, " \t").empty())
            continue;

        stringstream parseme(line);
        string keyword, symbol;
        parseme >> keyword;

        if (to_uppercase_string(keyword) == "BLANK_CHARACTER:")
        {
            if ((parseme >> symbol) && (symbol.length() == 1) && (symbol[0] != '\\')
&&
                (symbol[0] != '[') && (symbol[0] != ']') && (symbol[0] != '<') &&
                (symbol[0] != '>') && (symbol[0] >= '!') && (symbol[0] <= '~'))
            {
                blank_character = symbol[0];
                return;
            }
            else
            {
                cout << "ERROR: Illegal Blank Character. " << endl;
                valid = false;
                return;
            }
        }
    }

    cout << "ERROR: Missing or Incorrect BLANK_CHARACTER Keyword. " << endl;
    valid = false;
}

void tape::validate(const input_alphabet& input_alphabet,
                    const tape_alphabet& tape_alphabet,
                    bool& valid) const
{
    if (input_alphabet.is_element(blank_character))
    {
        cout << "Blank Character: " << blank_character << " is IN Input Alphabet. " <<
endl;
        valid = false;
    }

    if (!tape_alphabet.is_element(blank_character))
    {
        cout << "Blank Character: " << blank_character << " is NOT in Tape Alphabet."
<< endl;
        valid = false;
    }
}

void tape::view() const
{
    cout << "B = " << blank_character << "\n\n";
}

void tape::initialize(string input_string)
{
    cells = input_string + blank_character;
    current_cell = 0;
}

void tape::update(char write_character, direction move_direction)
{
    move_direction = toupper(move_direction);

    if ((move_direction == 'L') && (current_cell == 0))
        return;

    if ((move_direction == 'R') && (current_cell == cells.length() - 1))
    {
        cells += blank_character;
    }

    cells[current_cell] = write_character;

    if (move_direction == 'L')
        --current_cell;
    else
        ++current_cell;
}

string tape::left(int maximum_number_of_cells) const
{
    int first_cell = max(0, current_cell - maximum_number_of_cells);
    string value = cells.substr(first_cell, current_cell - first_cell);
    if (value.length() < static_cast<size_t>(current_cell))
        value.insert(0, "<");
    return value;
}

string tape::right(int maximum_number_of_cells) const
{
    int end_cell = cells.length() - 1;
    while ((end_cell >= current_cell) && (cells[end_cell] == blank_character))
        --end_cell;
    int last_cell = min(end_cell, current_cell + maximum_number_of_cells - 1);
    string value = cells.substr(current_cell, last_cell - current_cell + 1);

    if (value.length() < static_cast<size_t>(end_cell - current_cell + 1))
        value.append(">");
    return value;
}

char tape::current_character() const
{
    return cells[current_cell];
}

bool tape::is_first_cell() const
{
    return (current_cell == 0);
}

char tape::get_blank_character() const
{
    return blank_character;
}
```

## Tape.h:

```
//verified on 4/14
#ifndef tape_h
#define tape_h

#include "input_alphabet.h"
#include "tape_alphabet.h"
#include "direction.h"

#include <string>
#include <fstream>

using namespace std;

class tape
{
private:
    string cells;
    int current_cell;
    char blank_character;
public:
    tape();

    void load(ifstream& definition, bool& valid);

    void validate(const input_alphabet& input_alphabet,
                  const tape_alphabet& tape_alphabet,
                  bool& valid) const;

    void view() const;

    void initialize(string input_string);

    void update(char write_character,
                direction move_direction);

    string left(int maximum_number_of_cells) const;

    string right(int maximum_number_of_cells) const;

    char current_character() const;

    bool is_first_cell() const;

    char get_blank_character() const;

};

#endif
```

## Test.cpp

```
/*

#include "tape.h"
#include "direction.h"

#include <exception>
#include <string>
#include <iostream>


using namespace std;

int main()
{
    const int success(0);
    tape tape;
    tape.initialize("AABB");
    try
    {
        tape.update('B','L');
    }
    catch(const exception& error)
    {
        cerr << error.what() << '\n';
    }
    tape.view();
    return success;
}


*/
//ask which one it is

/*
#include "state.h"
#include <string>
#include <iostream>

using namespace std;

int main()
{
    const int success = 0;
    string name;
    state first_state;
    state second_state = "s2";
    state third_state = second_state;
    state_pointer fourth_state_pointer = new state(third_state);
    first_state = *fourth_state_pointer;
    fourth_state_pointer -> set_name("s3");
    delete fourth_state_pointer;
    first_state.get_name(name);

    cout << "Name of First State is " << name << endl;
    cout << "Total NUmber of States is " << state::total_number_of_states() << endl;

    return success;
}
*/
```

### Transition_function.cpp:

```
//final?
```

```
#include "transition_function.h"
#include "transition.h"
#include "tape_alphabet.h"
#include "states.h"
#include "final_states.h"
#include "direction.h"
#include "uppercase.h"
#include "string_play.h"

#include <string>
#include <vector>
#include <fstream>
#include <iostream>
#include <sstream>

using namespace std;

void transition_function::load(ifstream& definition, bool& valid)
{
    string line;
    string token;
    stringstream parseme;

    // Look for "TRANSITION_FUNCTION:"
    while (getline(definition, line))
    {
        if (trim(line, " \t").empty()) continue;

        parseme.clear();
        parseme.str(line);
        parseme >> token;

        if (uppercase_string(token) == "TRANSITION_FUNCTION:")
            break;
    }

    if (uppercase_string(token) != "TRANSITION_FUNCTION:")
    {
        cout << "ERROR: Transition Function Section Either; NOT defined OR Incorrect
Location. " << endl;
        valid = false;
        return;
    }

    // Load transitions
    while (getline(definition, line))
    {
        if (trim(line, " \t").empty())
```

```
            continue;

        parseme.clear();
        parseme.str(line);

        string source_state;
        char read_char;
        string destination_state;
        char write_char;
        string direction_token;

        if (!(parseme >> source_state >> read_char >> destination_state >> write_char
>> direction_token))
        {
            if (uppercase_string(source_state) == "INITIAL_STATE:")
            {
                definition.seekg(-(streamoff)line.length() - 1, ios::cur);
                break;
            }

            cout << "ERROR: Malformed Transition Line: " << line << endl;
            valid = false;
            continue;
        }

        // Validate direction character
        char dir_char = toupper(direction_token[0]);
        if (dir_char != 'L' && dir_char != 'R' && dir_char != 'S')
        {
            cout << "ERROR: Invalid Direction Symbol in Transition: " <<
direction_token << " (line: " << line << ")" << endl;;
            valid = false;
            continue;
        }

        try
        {
            transition t(source_state, read_char, destination_state, write_char,
dir_char);
            transitions.push_back(t);
        }
        catch (...)
        {
            cout << "ERROR: Exception Occurred While Creating Transition from Line: "
<< line << endl;
            valid = false;
        }
    }
```

```
}

void transition_function::validate(const tape_alphabet& tape_alphabet,
                                   const states& states,
                                   const final_states& final_states,
                                   bool& valid) const
{
    for (const auto& t : transitions)
    {
        if (final_states.is_element(t.source_state()))
        {
            cout << "Source State '" << t.source_state() << "' is IN final states. "
<< endl;
            valid = false;
        }
        if (!states.is_element(t.source_state()))
        {
            cout << "Source State '" << t.source_state() << "' is NOT in states. " <<
endl;
            valid = false;
        }
        if (!tape_alphabet.is_element(t.read_character()))
        {
            cout << "Read Character '" << t.read_character() << "' is NOT in tape
alphabet. " << endl;
            valid = false;
        }
        if (!states.is_element(t.destination_state()))
        {
            cout << "Destination State '" << t.destination_state() << "' is NOT in
states. " << endl;
            valid = false;
        }
        if (!tape_alphabet.is_element(t.write_character()))
        {
            cout << "Write Character '" << t.write_character() << "' is NOT in tape
alphabet. " << endl;
            valid = false;
        }
    }
}

void transition_function::view() const
{
    cout << "Transitions: " << endl;
    for (const auto& t : transitions)
    {
        cout << "δ(" << t.source_state() << ", " << t.read_character() << ") = ("
             << t.destination_state() << ", " << t.write_character() << ", " <<
t.move_direction() << ")" << endl;
    }
    cout << endl;
}

void transition_function::find_transition(string source_state,
                                          char read_character,
                                          string& destination_state,
                                          char& write_character,
                                          direction& move_direction,
                                          bool& found) const
{
    for (const auto& t : transitions)
    {
        if (t.source_state() == source_state && t.read_character() == read_character)
        {
            destination_state = t.destination_state();
            write_character = t.write_character();
            move_direction = t.move_direction();
            found = true;
            return;
        }
    }
    found = false;
}
```

## Transition_function.h:

```
//verified on 4/14
#ifndef transition_function_h
#define transition_function_h

#include "transition.h"
#include "tape_alphabet.h"
#include "states.h"
#include "final_states.h"
#include "direction.h"

#include <string>
#include <vector>
#include <fstream>

using namespace std;

class transition_function
{
    private:
        vector <transition> transitions;
    public:
        void load(ifstream& definition,
                    bool& valid);
        void validate( const tape_alphabet & tape_alphabet,
                        const states& states,
                        const final_states& final_states,
                        bool& valid) const;
        void view() const;
        void find_transition (string source_state,
                                char read_character,
                                string& destination_state,
                                char& write_character,
                                direction& move_direction,
                                bool& found) const;

};

#endif
```

## Transition.cpp:

```
//verified on 4/14
#include "transition.h"
#include "direction.h"

#include <string>

using namespace std;

transition::transition(string source_state,
                        char read_character,
                        string destination_state,
                        char write_character,
                        direction move_direction):

                        source(source_state),
                        read(read_character),
                        destination(destination_state),
                        write(write_character),
                        move(move_direction)
{

}

string transition::source_state() const
{
    return source;
}


char transition::read_character() const
{

    return read;
}

string transition::destination_state() const
{

    return destination;
}

char transition::write_character() const
{

    return write;
}

direction transition::move_direction() const
{

    return move;
}
```

## Transition.h:

```
//verified 4/14
#ifndef transition_h
#define transition_h

#include "direction.h"

#include <string>

using namespace std;

class transition
{
    private:
        string source;
        char read;
        string destination;
        char write;
        direction move;
    public:
        transition(string source_state,
                    char read_character,
                    string destination_state,
                    char write_character,
                    direction move_direction);
        string source_state() const;
        char read_character() const;
        string destination_state() const;
        char write_character() const;
        direction move_direction() const;

};

#endif
```

## Turing_machine.cpp:

```
//final?
#include "turing_machine.h"
#include "uppercase.h"
#include "crash.h"
#include "string_play.h"

#include <fstream>
#include <sstream>
#include <iostream>

turing_machine::turing_machine(std::string definition_file_name)
{
    used = false;
    valid = true;
    accepted = false;
    rejected = false;

    std::ifstream definition(definition_file_name.c_str());
    if (!definition)
    {
        std::cout << "Error: Could NOT Open: " << definition_file_name << std::endl;
        valid = false;
        return;
    }

    // Load each section
    definition.clear(); definition.seekg(0);
    states_data.load(definition, valid); if (!valid) return;

    definition.clear(); definition.seekg(0);
    input_alphabet_data.load(definition, valid); if (!valid) return;

    definition.clear(); definition.seekg(0);
    tape_alphabet_data.load(definition, valid); if (!valid) return;

    definition.clear(); definition.seekg(0);
    transition_function_data.load(definition, valid); if (!valid) return;

    // Load initial state
    std::string line, keyword;
    std::streampos pos = definition.tellg();
    while (std::getline(definition, line) && trim(line, " \t").empty());

    std::stringstream parse_line(line);
    parse_line >> keyword;

    if (uppercase_string(keyword) != "INITIAL_STATE:")
    {
        definition.seekg(pos, std::ios::beg);
        std::cout << "ERROR: Initial State NOT Defined. " << std::endl;
        valid = false;
        return;
    }
    else
    {
        parse_line >> initial_state;
    }

    definition.clear(); definition.seekg(0);
    tape_data.load(definition, valid); if (!valid) return;

    definition.clear(); definition.seekg(0);
    final_states_data.load(definition, valid); if (!valid) return;

    // Additional Validation
    if (valid)
    {
        if (!states_data.is_element(initial_state))
        {
            valid = false;
            std::cout << "Initial State '" << initial_state << "' is NOT in the Set-
of-States. " << std::endl;
        }

        if (input_alphabet_data.is_element(tape_data.get_blank_character()))
        {
            valid = false;
            std::cout << "Input Alphabet should NOT contain the Blank Character. " <<
std::endl;
        }

        if (!tape_alphabet_data.is_element(tape_data.get_blank_character()))
        {
            valid = false;
            std::cout << "Tape Alphabet MUST contain the Blank Character. "
<<std::endl;
        }

        transition_function_data.validate(tape_alphabet_data, states_data,
final_states_data, valid);
    }

    std::cout << std::endl;
    //std::cout << "Turing Machine " << definition_file_name << (valid ? "
Successfully Loaded.\n" : " Failed to Load Due to Errors. \n");

    definition.close();
}

void turing_machine::view_definition() const
{
    std::cout << "Definition of Turing Machine:\n";
    states_data.view();
    input_alphabet_data.view();
    tape_alphabet_data.view();
    transition_function_data.view();
    std::cout << "q0 = " << initial_state << "\n\n";
    tape_data.view();
    final_states_data.view();
}

void turing_machine::view_instantaneous_description(int max_cells) const
{
    std::cout << tape_data.left(max_cells)
              << "[" << current_state << "]"
              << tape_data.right(max_cells) << "\n\n";
}

bool turing_machine::initialize(std::string input)
{
    if (used)
    {
        std::cout << "You Must Terminate the Currently Running Machine to Start a New 
One " << std::endl;
        std::cout << std::endl;
        return false;
    }

    original_input_string = input;
    current_state = initial_state;
    tape_data.initialize(input);

    number_of_transitions = 0;
    operating = true;
    used = true;
    accepted = false;
    rejected = false;

    return true;
}

void turing_machine::perform_transitions(int max_transitions)
{
    if (final_states_data.is_element(current_state))
    {
        operating = false;
        accepted = true;
        used = false;
        return;
    }

    std::string next_state;
    char write_char;
    direction move_dir;
    bool found = false;

    for (int i = 0; i < max_transitions && operating; i++)
    {
        transition_function_data.find_transition(
            current_state,
            tape_data.current_character(),
            next_state,
            write_char,
            move_dir,
            found
        );

        if (!found)
        {
            rejected = true;
```

```cpp
            operating = false;
            used = false;
            return;
        }

        number_of_transitions++;
        tape_data.update(write_char, move_dir);
        current_state = next_state;

        if (final_states_data.is_element(current_state))
        {
            operating = false;
            accepted = true;
            used = false;
            return;
        }
    }
}

void turing_machine::terminate_operation()
{
    if (operating)
    {
        operating = false;
        used = false;
        accepted = false;
        rejected = false;
    }
}

std::string turing_machine::input_string() const
{
    return original_input_string;
}

int turing_machine::total_number_of_transitions() const
{
    return number_of_transitions;
}

bool turing_machine::is_valid_definition() const
{
    return valid;
}

bool turing_machine::is_valid_input_string(std::string value) const
{
    for (char ch : value)
    {
        if (!input_alphabet_data.is_element(ch))
            return false;
    }
    return true;
}

bool turing_machine::is_used() const
{
    return used;
}

bool turing_machine::is_operating() const
{
    return operating;
}

bool turing_machine::is_accepted_input_string() const
{
    return accepted;
}

bool turing_machine::is_rejected_input_string() const
{
    return rejected;
}
```

## Turing_machine.h:

```
//verified on 4/14
#ifndef turing_machine_h
#define turing_machine_h

#include "tape.h"
#include "input_alphabet.h"
#include "tape_alphabet.h"
#include "transition_function.h"
#include "states.h"
#include "final_states.h"

#include <string>
#include <vector>

using namespace std;

class turing_machine
{
```

```
private:
    tape tape_data;
    input_alphabet input_alphabet_data;
    tape_alphabet tape_alphabet_data;
    transition_function transition_function_data;
    states states_data;
    final_states final_states_data;
    vector<string> description;
    string initial_state;
    string current_state;
    string original_input_string;
    int number_of_transitions;
    bool valid;
    bool used;
    bool operating;
    bool accepted;
    bool rejected;

public:
    turing_machine(string definition_file_name);

    void view_definition() const;
    void view_instantaneous_description(int maximum_number_of_cells) const;
    bool initialize(string input_string);
    void perform_transitions(int maximum_number_of_transitions);
    void terminate_operation();

    string input_string() const;
    int total_number_of_transitions() const;
    bool is_valid_definition() const;
    bool is_valid_input_string(string value) const;
    bool is_used() const;
    bool is_operating() const;
    bool is_accepted_input_string() const;
    bool is_rejected_input_string() const;
};

#endif
```

## Uppercase.h:

```
//final?
#ifndef uppercase_h
#define uppercase_h

#include <string>
```

```
#include <algorithm>

// Core implementation
inline std::string to_uppercase_string(std::string value)
{
    std::transform(value.begin(), value.end(), value.begin(), ::toupper);
    return value;
}

// Aliases for compatibility
inline std::string uppercase_string(std::string value) { return
to_uppercase_string(value); }

#endif
```

## Visible.cpp:

```
//final?
#include "visible.h"
#include <string>
using namespace std;

// makes empty strings that are the strings to run on visible
string visible(string value)
{
    const string lambda("\\");
    if(value.empty())
        value=lambda;
    return value;
}
```

## Visible.h:

```
//final?
#ifndef visible_h
#define visible_h

#include <string>

using namespace std;

string visible(string value);

#endif
```