REPORT

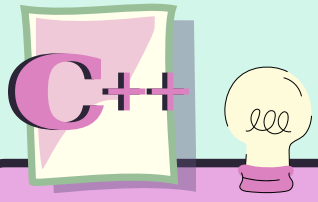
Angela Paola Lozano Ochoa

TABLE OF CONTENTS

contents

Section 1.

The essence of your solution



For the implementation of the Posfix++ operator, C++ was utilized due to its suitability for object-oriented projects, allowing for encapsulation of various structures in classes. Additionally, C++ supports template classes, enabling the creation of data structures that work with any data type, enhancing flexibility and reusability.

Initially, a fixed-size stack array-based implementation was developed using an array, ensuring efficient management of stack operations. Following this, a hash table was implemented to handle variable assignments, limited to 26 letters (A-Z). This hash table uses a hashing function to determine the appropriate array index for storing key-value pairs.

Lastly, an evaluator was implemented to process all postfix expressions and perform the respective operations. Each component was encapsulated in classes to maintain a modular and organized code structure. Proper error handling was incorporated to manage expression processing errors. Unit tests were also implemented to ensure the program's functionality. The current program supports 29 distinct mathematical operations and one assignment operation.

Section 2.

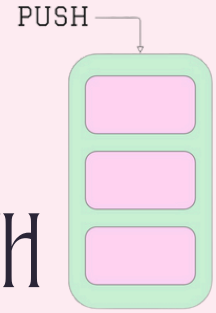
Explanation of the original algorithms of the solution

Stack Implementation

The solution includes the implementation of a fixed-size stack using an array. This stack is responsible for storing intermediate results and operands during the evaluation of the postfix expression.

OPERATIONS - PUSH

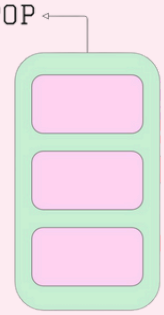
PUSH: Adds an element to the top of the stack. Before adding, it checks if the stack has reached its maximum capacity. If the stack is full, it dynamically resizes to accommodate more elements, preventing overflow.



PUSH

OPERATIONS - POP

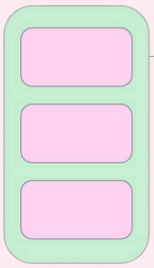
POP: Removes the top element from the stack. It checks if the stack is empty to avoid errors. If the stack is empty, it triggers an underflow error. Otherwise, it retrieves and removes the top element, decreasing the stack size by one.



POP

OPERATIONS - PEEK

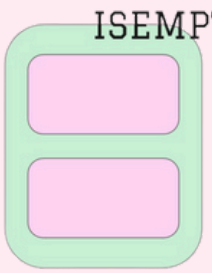
PEEK: Inspects the top element of the stack without removing it. This operation checks if the stack is empty to avoid errors, allowing the current state of the stack to be examined safely.



PEEK

OPERATIONS - ISEMPTY

ISEMPTY: Checks if the stack contains any elements, returning a boolean value. This operation helps control the flow of the postfix evaluation process by determining if the stack has any elements left.

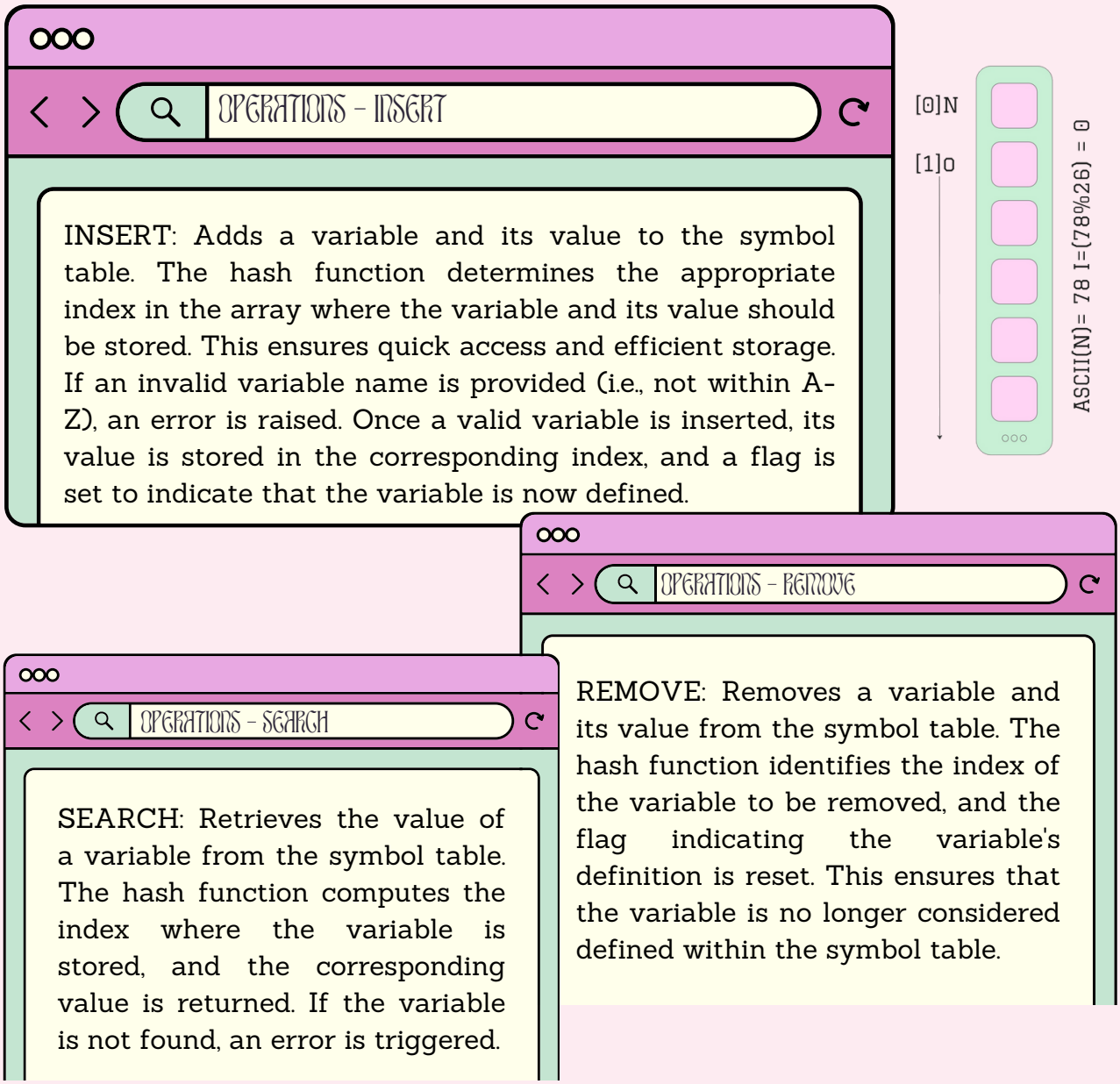


ISEMPTY

Symbol Table Implementation

The solution implements a symbol table using a hash table to store variable values, accommodating 26 letters (A-Z). This choice of data structure optimizes the search and retrieval process for variable values, making it both efficient and straightforward.

The hash table uses a hash function that calculates the index for each letter by taking the modulo 26 of the ASCII value of the letter. This ensures that each letter is mapped to a unique index in the range of 0 to 25, corresponding to the 26 letters of the alphabet. This approach eliminates the need for a linear search, reducing the time complexity of lookup operations to constant time, O(1).

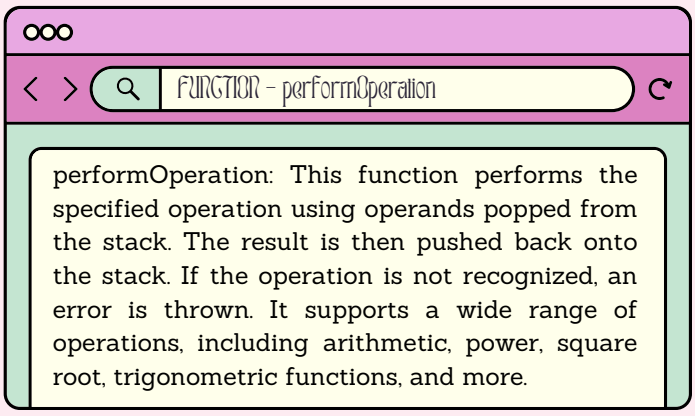


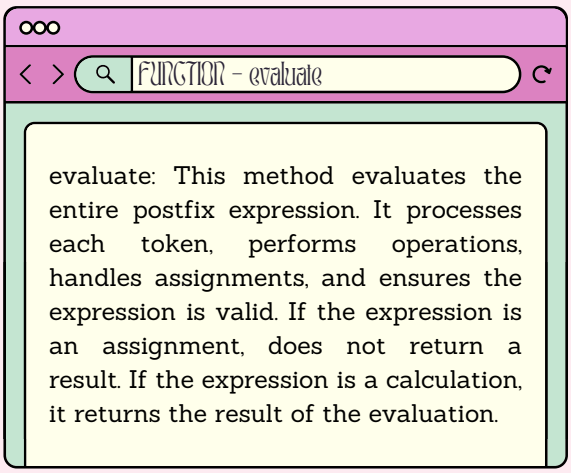
Postfix Expression Evaluator

The postfix expression evaluator processes postfix expressions to perform mathematical calculations and variable assignments. It utilizes the created stack data structure and symbol table.

The evaluator reads the postfix expression token by token. When encountering a number or a variable, the token is pushed onto the stack. When an operator is encountered, the evaluator pops the required number of operands from the stack (dependent on the operator) and performs the corresponding operation. The result of this operation is then pushed back onto the stack and then is finally popped to be returned. If the operator is an assignment, it does not push a result back onto the stack. Instead, it assigns the value to a variable in the symbol table.
For example, in the expression A 5 =, the evaluator assigns the value 5 to the variable A without returning a result. The evaluator ensures that the expression is coherent. If the expression is invalid, such as 5 6 4 *, the evaluator identifies this and reports an error.

The operations supported by the evaluator include basic arithmetic operations (addition, subtraction, multiplication, division), advanced mathematical functions (square root, logarithms, trigonometric functions), and an assignment operation. There are 29 different mathematical operations available in total.





Section 3.

Pseudocode

Stack

stack | symbolTable | operations | postfix

```
INITIAL_CAPACITY = 10; // Maximum size of the stack
T *items;              // Array to store stack elements
int top;               // Index of the top element
int capacity;          // Actual capacity of the stack

//Initializes the stack indicating the stack is empty.
STACK_INITIALIZE()
    top ← -1
    capacity ← INITIAL_CAPACITY
END STACK_INITIALIZE

/**
 * Resize the array.
 */
RESIZE()
  T temp = new T[capacity * 2];
  for (int i = 0; i < top; i++)
    {temp[i] = items[i]}
  items = temp; capacity *= 2;
END RESIZE

/**
 * Pushes a value onto the stack.
 * @param value The value to be pushed onto the stack.
 * @throws OverflowError if the stack is full.
 */
PUSH(value)
    IF (top ≥ capacity - 1) RESIZE()
    top ← top + 1;  items[top] ← value;
END PUSH


/**
 * Pops the top value from the stack.
 * @return The value at the top of the stack.
 * @throws UnderflowError if the stack is empty.
 */
POP()
    IF top < 0 THEN
        THROW "Stack is empty: cannot pop."
    END IF
    value ← items[top]; top ← top - 1;
    RETURN value
END POP


/**
 * Peeks at the top value of the stack without removing it.
 * @return The value at the top of the stack.
 * @throws UnderflowError if the stack is empty.
 */
PEEK()
    IF top < 0 THEN
        THROW "Stack is empty: cannot peek."
    END IF
    RETURN items[top]
END PEEK

//Checks if the stack is empty.
IS_EMPTY()
    RETURN top = -1
END IS_EMPTY
```

Stack - Time Complexity

STACK_INITIALIZE()
Time Complexity: 0(1)
Explanation: Initializing the top index
to -1 and setting the capacity to the
initial capacity are constant-time operations.

RESIZE()
Time Complexity: 0(n)
Explanation: Creating a new array of double
the capacity and copying all elements from
the old array to the new array takes linear time,
where n is the number of elements in the stack.

PUSH(value)
Time Complexity: 0(1) (amortized)
Explanation: In the worst case, when the stack is
full, RESIZE() is called, which takes O(n) time.
However, due to the doubling strategy of the
resizing operation, the amortized cost of PUSH
operations over a series of operations is 0(1).

POP(): Time Complexity: 0(1)
PEEK(): Time Complexity: 0(1)
IS_EMPTY(): Time Complexity: 0(1)

CLEAR_STACK(): Time Complexity: 0(1)
Explanation: Resetting the top index to -1 is a
constant-time operation.

Symbol Table

stack | symbolTable

```
CONSTANT MAX_SIZE ← 26 //maximum size of the table
DECLARE table[0..MAX_SIZE-1] OF DOUBLE
// Array to store the values associated with the keys.
DECLARE isSet[0..MAX_SIZE-1] OF BOOLEAN
//Array to indicate whether a value has been set for each key

/**
 * Initializes the symbol table by setting all entries as unset.
 */
SYMBOL_TABLE_INITIALIZE()
    FOR i ← 0 TO MAX_SIZE - 1 DO
        isSet[i] ← false
    END FOR
END SYMBOL_TABLE_INITIALIZE


/**
 * Hash function to calculate the index for a given key.
 * @param key The character key to be hashed.
 * @return The index corresponding to the key.
 */
HASH(key)
    RETURN ASCII(key) MOD MAX_SIZE
END HASH


/**
 * Checks if the given key is valid.
 * @param key The character key to be validated.
 * @return True if the key is between 'A' and 'Z',
 false otherwise.
 */
IS_VALID_KEY(key)
    IF (key is a letter between "A" AND "Z") THEN
        return TRUE
    ELSE return FALSE
END IS_VALID_KEY


/**
 * Inserts a key-value pair into the symbol table.
 * @param key The character key to be inserted.
 * @param value The value associated with the key.
 * @throws OutOfRange if the key is invalid.
 */
INSERT(key, value)
    IF NOT IS_VALID_KEY(key) THEN
        THROW "Invalid key: must be between 'A' and 'Z'."
    END IF
    index ← HASH(key)
    table[index] ← value
    isSet[index] ← true
END INSERT


/**
 * Searches for a value associated with a given key in the
 symbol table.
```

```
 * @param key The character key to be searched.
 * @return The value associated with the key.
 * @throws OutOfRange if the key is invalid or not set.
 */
SEARCH(key)
    IF NOT IS_VALID_KEY(key) THEN
        THROW "Invalid key: must be between 'A' and 'Z'."
    END IF
    index ← HASH(key)
    IF NOT isSet[index] THEN
        THROW "Key not found: the specified key is not set."
    END IF
    RETURN table[index]
END SEARCH


/**
 * Removes a key-value pair from the symbol table.
 * @param key The character key to be removed.
 * @throws OutOfRange if the key is invalid or not set.
 */
REMOVE(key)
    IF NOT IS_VALID_KEY(key) THEN
        THROW "Invalid key: must be between 'A' and 'Z'."
    END IF
    index ← HASH(key)
    IF NOT isSet[index] THEN
        THROW "Key not found: the specified key is not set."
    END IF
    isSet[index] ← false
END REMOVE
```

Symbol Table - Time Complexity

SYMBOL_TABLE_INITIALIZE()
Time Complexity: O(1)
Explanation: Setting all entries in the isSet array to false for a fixed size of MAX_SIZE is a constant time operation since MAX_SIZE is a constant (26).

HASH(key)
Time Complexity: O(1)
Explanation: Calculating the hash index using the ASCII value of the key and taking the modulo with MAX_SIZE is a constant-time operation.

IS_VALID_KEY(key) Time Complexity: O(1)
INSERT(key, value) Time Complexity: O(1)

SEARCH(key)
Time Complexity: O(1)
Explanation: Validating the key, calculating the hash index, and retrieving the value from the table (or checking if it is set) are all constant time operations.

REMOVE(key)
Time Complexity: O(1)
Explanation: Validating the key, calculating the hash index, and setting the corresponding isSet entry to false are all constant-time operations.

Operations

stack | symbolTable | operations

```
FUNCTION ADD(a, b)
    return a + b
END FUNCTION

FUNCTION SUBTRACT(a, b)
    return a - b
END FUNCTION

FUNCTION MULTIPLY(a, b)
    return a * b
END FUNCTION

FUNCTION DIVIDE(a, b)
    if b = 0 then
        THROW "Division by
    end if
    return a / b
END FUNCTION

FUNCTION POWER(a, b)
    return a ^ b
END FUNCTION

FUNCTION MODULUS(a, b)
    return a MOD b
END FUNCTION

FUNCTION SQUARE_ROOT(a)
    return √a
END FUNCTION

FUNCTION CUBE_ROOT(a)
    return a^(1/3)
END FUNCTION

FUNCTION HYPOTENUSE(x, y)
    return √(x^2 + y^2)
END FUNCTION

FUNCTION FLOOR(a)
    return ⌊a⌋
END FUNCTION

FUNCTION CEIL(a)
    return ⌈a⌉
END FUNCTION

FUNCTION ABS(a)
    return |a|
END FUNCTION

FUNCTION SIGN(a)
    if a > 0 then
        return 1
    else if a < 0 then
        return -1
    else
        return 0
    end if
END FUNCTION
```

```
FUNCTION ROUND(a)
    return ROUND(a)
END FUNCTION

FUNCTION FACTORIAL(n)
    if n < 0 then
        THROW "Factorial is not
          defined for negative numbers"
    else if n = 0 OR n = 1 then
        return 1
    else
        return n * FACTORIAL(n - 1)
    end if
END FUNCTION

FUNCTION LOG10(a)
    return LOG10(a)
END FUNCTION

FUNCTION LOG2(a)
    return LOG2(a)
END FUNCTION

FUNCTION PERM(n, k)
    if n < 0 OR k < 0 OR k > n then
        THROW "Invalid values for perm(n, k)"
    end if
    return FACTORIAL(n) / FACTORIAL(n - k)
END FUNCTION

FUNCTION COMB(n, k)
    if n < 0 OR k < 0 OR k > n then
        THROW "Invalid values for comb(n, k)"
    end if
    return FACTORIAL(n) /
    (FACTORIAL(k) * FACTORIAL(n - k))
END FUNCTION

FUNCTION GCD(x, y)
    if y = 0 then
        return ABS(x)
    else
        return GCD(y, x MOD y)
    end if
END FUNCTION

FUNCTION LCM(x, y)
    if x = 0 OR y = 0 then
        return 0
    else
        return ABS(x * y) / GCD(x, y)
    end if
END FUNCTION

FUNCTION SIN(a)
    return SIN(a)
END FUNCTION

FUNCTION COS(a)
    return COS(a)
END FUNCTION

FUNCTION TAN(a)
    return TAN(a)
END FUNCTION
```

Postfix Evaluator


stack
symbolTable
operations
postfix
index.ts
else if (operation == "abs"):
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.ABS(a)))
else if (operation == "sgn"):
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.SIGN(a)))
else if (operation == "round"):
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.ROUND(a)))
else if (operation == "!"):
a ← TO_INTEGER(EVALUATE_VALUE(stack.POP()))
stack.PUSH(TO_STRING(MathOperations.FACTORIAL(a)))
else if (operation == "log10"):
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.LOG10(a)))
else if (operation == "log2"):
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.LOG2(a)))
else if (operation == "perm"):
b ← EVALUATE_VALUE(stack.POP())
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.PERM(a, b)))
else if (operation == "comb"):
b ← EVALUATE_VALUE(stack.POP())
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.COMB(a, b)))
else if (operation == "gcd"):
b ← EVALUATE_VALUE(stack.POP())
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.GCD(a, b)))
else if (operation == "lcm"):
b ← EVALUATE_VALUE(stack.POP())
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.LCM(a, b)))
else if (operation == "sin"):
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.SIN(a)))
else if (operation == "cos"):
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.COS(a)))
else if (operation == "tan"):
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.TAN(a)))
else if (operation == "asin"):
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.ARC_SIN(a)))
else if (operation == "acos"):
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.ARC_COS(a)))
else if (operation == "atan"):
a ← EVALUATE_VALUE(stack.POP())
stack.PUSH(TO_STRING(MathOperations.ARC_TAN(a)))
else
THROW "Invalid token in expression: " + operation
END IF
END FUNCTION


stack
symbolTable
operations
postfix
/**
* Evaluates a postfix expression.
* @param expression The postfix expression to evaluate.
* @return The result of the evaluation.
* @throws Runtime error if the expression is invalid.
*/
FUNCTION EVALUATE(expression)
DECLARE tokens AS LIST OF STRING = SPLIT(expression, " ")
DECLARE assign AS BOOL
FOR EACH token IN tokens DO
IF IS_NUMBER(token) OR IS_VARIABLE(token) THEN
stack.PUSH(token)
ELSE IF token = "=" THEN
DECLARE value AS DOUBLE
= EVALUATE_VALUE(stack.POP())
DECLARE key AS STRING = stack.POP()
IF LENGTH(key) ≠ 1 OR NOT IS_ALPHA(key) THEN
THROW "Invalid variable name"
END IF
SYMBOL_TABLE.INSERT(key, value)
assign = true
ELSE
PERFORM_OPERATION(token)
END IF
END FOR
DECLARE result as NULL
IF stack.is_empty() = FALSE
result = stack.pop()
if stack.is_empty() = FALSE THEN
THROW "Invalid expression"
END IF
RETURN result
END FUNCTION

Evaluator - Time Complexity

EVALUATE_VALUE(token)
Time Complexity: O(1)
Explanation: Checking if a token is a variable and retrieving its value from the symbol table or converting the token to a number are all constant time operations.

IS_NUMBER(token) Time Complexity: O(1)
IS_VARIABLE(token) Time Complexity: O(1)

PERFORM_OPERATION(operation)
Time Complexity: O(1)
Explanation: Validating the operation and performing arithmetic or mathematical operations, including popping from and pushing to the stack, are all constant-time operations. Each individual operation (addition, subtraction, etc.) is assumed to be O(1).

EVALUATE(expression)
Time Complexity: O(n)
Explanation: Splitting the expression into tokens takes O(n) time, where n is the number of characters in the expression. Iterating over each token and performing operations based on the token type (number, variable, or operator) involves operations that are each O(1). The overall complexity is linear with respect to the length of the expression.

Section 4.

Data structures

Stack - Data Structures

Stack - Data structure Explanation I

In the implementation of the stack, a *dynamic array* is utilized with an initial capacity of 10 spaces, which can be resized as needed.
Initially, a stack was implemented using a fixed-size array implementation std::array<typename, size>. However, with the aim of providing the stack with more advanced and scalable capabilities, it was deemed necessary to address the efficient management of element storage when the stack reached its maximum capacity. To achieve this, a migration to a dynamic array approach typename* was chosen. The adopted approach, based on a dynamic array typename*, allows the stack to resize to double its current size whenever it becomes full. When the maximum capacity of the array is reached, a new array with double the capacity is dynamically allocated, the existing elements are copied to the new array, and the memory of the original array is released. Although this involves a copy operation during resizing, the ability to grow dynamically as needed avoids the necessity to predict the maximum stack size from the beginning, optimizing memory usage and improving overall performance compared to a fixed-size array. Additionally, by doubling the size with each resize, the frequency of copy operations is minimized, contributing to better performance in scenarios where frequent or large numbers of stack operations are performed. It is important to highlight that this implementation forgoes any predefined methods provided by standard libraries, such as .pop() or .push(), opting instead for a completely custom and optimized approach tailored to the specific needs of a stack. Every operation, from insertion to element removal, was developed from scratch."

Stack - Data structure Explanation II
Benefits of using an array

Firstly, employing a dynamic array facilitates efficient access and management of the stack elements. The stack is managed through a pointer that consistently points to the top of the stack. This pointer aids in expeditiously determining whether the stack is empty or in accessing the topmost element.

Secondly, dynamic arrays offer constant time complexity, O(1), for accessing and updating elements. This characteristic is crucial for the push and pop operations, which necessitate swift execution.

Moreover, the use of an array allows for easy iteration, which is pivotal for the functionality of the pop operation. When an element is popped from the stack, the pointer is decremented to simulate the removal of the top element.

Within the stack, string values are stored. This decision is pertinent for languages like C++, where strict type checking can cause errors if the stack were to store mixed types [1]. By utilizing strings, the stack can handle expressions such as 'A 5 =', where 'A' and '5' are of different types. Storing everything as strings ensures that the stack can process variable names and numbers without type conflicts.

Symbol Table - Data Structures

Symbol Table - Data structure Explanation

In the implementation of the symbol table, two static arrays are employed. This design decision is driven by several factors. Firstly, the utilization of static arrays provides a straightforward and efficient means to store and access values. One array stores the values assigned to each key, while the other array, a boolean array, indicates whether a particular key is set. Both arrays have a fixed size of 26, corresponding to the 26 uppercase letters A-Z.

DETAILED EXPLANATION:

The *primary array*, referred to as the *table*, stores the double values associated with each key. This array has a fixed size of 26, providing a direct mapping from each key to its corresponding value via the hash function. The fixed size ensures that each key has a predefined slot, eliminating the need for dynamic resizing or complex memory management. The *secondary array*, referred to as *isSet*, is a boolean array of the same size (26). Each element in this array indicates whether the corresponding slot in the table array has been set with a value. This is essential for efficient lookups and deletions, as it allows the program to quickly determine if a key has been previously assigned a value.

Extra

Extra

The final evaluation of the postfix expression involves iterating through each token of the expression using *std::istringstream iss(expression)*, which tokenizes the input string based on whitespace. Each token is then processed to determine whether it is a number, a variable, or an operator.

If the token is identified as a number or a variable, it is pushed onto the stack as a string. This approach maintains the flexibility to handle different data types without causing type conflicts. When an operator token is encountered, it triggers the evaluation process. The evaluator recognizes the operation and pops the necessary operands from the stack. These operands, initially stored as strings, are converted to double values using *std::stod* for accurate numerical computation.

The evaluator then performs the specified operation using the converted double values. For operations that require integer inputs, the double values are cast to int using *static_cast<int>* to ensure the correct operation is applied. After the operation is performed, the result is converted back to a string using *std::to_string* and pushed onto the stack. This method ensures that the stack always contains string representations of the data, simplifying the handling of different data types.

If the operator token is an assignment operator *(=)*, the evaluator does not push the result back onto the stack. Instead, it processes the assignment by popping the value and the variable name from the stack, converting the value as necessary, and storing it in the symbol table.

Section 5.

Source code & Video

Stack I

stack.hpp stack.tpp

```hpp
/**
 * @file Stack.hpp
 * @brief This file contains the definition of the Stack class template.
 *
 * The Stack class template is used to manage a stack of elements of any
 type. It provides methods to push, pop, and peek at the top element, as
 well as to check if the stack is empty.
 */

#ifndef STACK_HPP
#define STACK_HPP

#include <iostream>
#include <stdexcept>

/**
 * @class Stack
 * @brief A class template to manage a stack of elements of any type.
 *
 * The Stack class template provides methods to manipulate a stack of
 elements. It uses an array to store the stack elements and keeps track
 of the index of the top element.
 * @tparam T The type of the elements in the stack.
 */
template <typename T>
class Stack
{
private:
    static const int INITIAL_CAPACITY = 10; // Maximum size of the stack
    T *items;                               // Array to store stack elements
    int top;                                // Index of the top element
    int capacity;                           // Actual capacity of the stack

    void resize(std::size_t newSize); // Resize the stack

public:
    /**
     * @brief Construct a new Stack object.
     */
    Stack();
    /**
     * @brief Destroy the Stack object and free allocated memory.
     */
    ~Stack();

    /**
     * @brief Push an element onto the stack.
     * @param[in] value The element to be pushed onto the stack.
     */
    void push(T value);

    /**
     * @brief Pop an element from the stack.
     * @return The element that was popped from the stack.
     * @throws std::underflow_error If the stack is already empty.
     */
    T pop();

    /**
     * @brief Get the top element of the stack without popping it.
     * @return The top element of the stack.
     * @throws std::underflow_error If the stack is already empty.
     */
    T peek() const;

    /**
     * @brief Check if the stack is empty.
     * @return True if the stack is empty, false otherwise.
     */
    bool isEmpty() const;
};

#include "Stack.tpp"

#endif // STACK_HPP
```

Stack II

Click here to view commented pseudocode

Pseudocode

```
STACK_INITIALIZE()
    top ← -1
    capacity ← INITIAL_CAPACITY
END STACK_INITIALIZE



RESIZE()
    T temp = new T[capacity*2]
    for (int i = 0; i < top; i++)
    {temp[i] = items[i]}
    items = temp
    capacity = capacity * 2
END RESIZE



PUSH(value)
    PUSH(value)
        IF (top ≥ capacity - 1)
          RESIZE()
        top ← top + 1
        items[top] ← value
    END PUSH
END PUSH



POP()
    IF top < 0 THEN
        THROW "Stack is empty:
          cannot pop."
    END IF
    value ← items[top]
    top ← top - 1
    RETURN value
END POP



PEEK()
    IF top < 0 THEN
        THROW "Stack is empty:
          cannot peek."
    END IF
    RETURN items[top]
END PEEK



IS_EMPTY()
    RETURN top = -1
END IS_EMPTY



CLEAR_STACK()
    top ← - 1
END CLEAR_STACK
```

stack.hpp | stack.tpp

```cpp
/**
 * @file Stack.tpp
 * @brief This file contains the implementation
 of the Stack class template.
 */

#include "Stack.hpp"
// Constructor initializes the stack
template <typename T>
Stack<T>::Stack() : top(-1), capacity(INITIAL_CAPACITY)
{
    items = new T[INITIAL_CAPACITY];
}

template <typename T>
Stack<T>::~Stack()
{
    delete[] items;
}

template <typename T>
void Stack<T>::resize(std::size_t newSize)
{
    T *temp = new T[newSize];
    for (int i = 0; i < top; i++)
    {
        temp[i] = items[i];
    }
    delete[] items;
    items = temp;
    capacity = newSize;
}

template <typename T>
void Stack<T>::push(T value)
{
    if (top >= capacity - 1)
    {

        resize(capacity * 2);
    }
    items[++top] = value;
}
template <typename T>
T Stack<T>::pop()
{
    if (top < 0)
    {
        throw std::underflow_error("Stack is empty: cannot pop.");
    }
    return items[top--];
}
template <typename T>
T Stack<T>::peek() const
{
    if (top < 0)
    {
        throw std::underflow_error("Stack is empty: cannot peek.");
    }
    return items[top];
}
template <typename T>
bool Stack<T>::isEmpty() const
{
    return top == -1;
}
```

Stack - test

test_stack.cpp

```cpp
#include "../src/stack.hpp"
#include "gtest/gtest.h"

class StackTest : public testing::Test
{
protected:
    Stack<int> stack;

    void SetUp() override
    {
        stack.push(1);
        stack.push(2);
    }

    void TearDown() override
    {
        while (!stack.isEmpty())
        {
            stack.pop();
        }
    }
};


TEST_F(StackTest, PushPop)
{
    EXPECT_EQ(stack.pop(), 2);
    EXPECT_EQ(stack.pop(), 1);
}

TEST_F(StackTest, Peek)
{
    EXPECT_EQ(stack.peek(), 2);
}

TEST_F(StackTest, IsEmpty)
{
    EXPECT_FALSE(stack.isEmpty());
    stack.pop();
    stack.pop();
    EXPECT_TRUE(stack.isEmpty());
}

TEST_F(StackTest, Underflow)
{
    stack.pop();
    stack.pop();
    EXPECT_THROW(stack.pop(), std::underflow_error);
    EXPECT_THROW(stack.peek(), std::underflow_error);
}

int main(int argc, char **argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Symbol Table I

stack.hpp | stack.tpp | symbol_table.hpp | symbol_table.hpp

```cpp
#ifndef SYMBOL_TABLE_HPP
#define SYMBOL_TABLE_HPP

/**
 * @file SymbolTable.hpp
 * @brief Implementation of a symbol table using a static array with a
 maximum size of 26 characters, representing the 26 letters of the alphabet.
 * @details This implementation uses the ASCII value of the letters to map
 them to their respective indices in the array, thus limiting the table to
 26 positions.
 */
#include <stdexcept>
#include <iostream>
#include <iomanip>
#include <array>

/**
 * @class SymbolTable
 * @brief A simple symbol table i
 * @details This class provides m
 key-value pairs from the symbol
 */
class SymbolTable
{
private:
    const std::string SOFT_PINK = "\033[38;5;225m";
    static const int MAX_SIZE = 26;
    // Maximum size of the symbol table
    std::array<double, MAX_SIZE> table = {};
    // Array to store the values associated with the keys
    std::array<bool, MAX_SIZE> isSet = {false};
    // Array to check if a key is set

    /**
     * @brief Calculates the hash value (index) for a given key.
     * @param key The character to calculate the hash value for.
     * @return The hash value (index) for the given key.
     */
    int hash(char key) const;

    /**
     * @brief Checks if the given key is a valid character.
     * @param key The character to check.
     * @return True if the key is a valid character, false otherwise.
     */
    bool isValidKey(char key) const;

public:
    /* *@brief Constructs an empty symbol table. */
    SymbolTable() = default;

    /**
     * @brief Inserts a key-value pair into the symbol table.
     * @param key The character representing the key.
     * @param value The value associated with the key.
     * @throws std::invalid_argument If the key is not a valid character.
     */
    void insert(char key, double value);

    /**
     * @brief Searches for a key in the symbol table and returns its
     associated value.
     * @param key The character representing the key to search for.
     * @return The value associated with the key, or 0.0 if the key is
     not found.
     * @throws std::invalid_argument If the key is not a valid character.
     */
    double search(char key) const;

    /**
     * @brief Removes a key-value pair from the symbol table.
     * @param key The character representing the key to remove.
     * @throws std::invalid_argument If the key is not a valid character.
     */
    void remove(char key);

    /**
     * @brief Prints the contents of the symbol table.
     * This method prints the contents of the symbol table (just assigned
     keys), including the keys and their associated values.
     * @details The method iterates through the symbol table and prints
     each key-value pair in the format:  - Key: <key> = Value: <value>
     */
    void printST() const;
};

#endif // SYMBOL_TABLE_HPP
```

Pseudocode

```
CONSTANT MAX_SIZE ← 26
DECLARE table[0..MAX_SIZE-1] OF DOUBLE
DECLARE isSet[0..MAX_SIZE-1] OF BOOLEAN

SYMBOL_TABLE_INITIALIZE()
    FOR i ← 0 TO MAX_SIZE - 1 DO
        isSet[i] ← false
    END FOR
END SYMBOL_TABLE_INITIALIZE
```

Symbol Table II

stack.hpp | stack.tpp | symbol_table.hpp | symbol_table.cpp

```cpp
#include "symbol_table.hpp"

// Hash function to calculate the index for a given key
int SymbolTable::hash(char key) const
{
    return (key - 'A') % MAX_SIZE;
}

bool SymbolTable::isValidKey(char key) const
{
    return (key >= 'A' && key <= 'Z');
}

void SymbolTable::insert(char key, double value)
{
    if (!isValidKey(key))
    {
        throw std::out_of_range("Invalid key:
          must be between 'A' and 'Z'.");
    }
    int index = hash(key);
    table[index] = value;
    isSet[index] = true;
}

double SymbolTable::search(char key) const
{
    if (!isValidKey(key))
    {
        throw std::out_of_range("Invalid key:
          must be between 'A' and 'Z'.");
    }
    int index = hash(key);
    if (!isSet[index])
    {
        throw std::out_of_range("Key not found:
          the specified key is not set.");
    }
    return table[index];
}

void SymbolTable::remove(char key)
{
    if (!isValidKey(key))
    {
        throw std::out_of_range("Invalid key:
          must be between 'A' and 'Z'.");
    }
    int index = hash(key);
    if (!isSet[index])
    {
        throw std::out_of_range("Key not found:
          the specified key is not set.");
    }
    isSet[index] = false;
}

void SymbolTable::printST() const
{
    std::cout << SOFT_PINK << "Symbol Table" << std::endl;
    std::cout << "------------" << std::endl;

    for (int i = 0; i < MAX_SIZE; ++i)
    {
        if (isSet[i])
        {
            std::cout << (char)('A' + i) << " = " <<
              table[i] << std::endl;
        }
    }
}
```

Pseudocode

```
HASH(key)
    RETURN ASCII(key) MOD MAX_SIZE
END HASH


IS_VALID_KEY(key)
    IF (key is a letter between
    "A" AND "Z") THEN
        return TRUE
    ELSE return FALSE
END IS_VALID_KEY


INSERT(key, value)
    IF NOT IS_VALID_KEY(key) THEN
        THROW "Invalid key: must be
        between 'A' and 'Z'."
    END IF
    index ← HASH(key)
    table[index] ← value
    isSet[index] ← true
END INSERT


SEARCH(key)
    IF NOT IS_VALID_KEY(key) THEN
        THROW "Invalid key: must be
        between 'A' and 'Z'."
    END IF
    index ← HASH(key)
    IF NOT isSet[index] THEN
        THROW "Key not found: the
        specified key is not set."
    END IF
    RETURN table[index]
END SEARCH


REMOVE(key)
    IF NOT IS_VALID_KEY(key) THEN
        THROW "Invalid key: must be
        between 'A' and 'Z'."
    END IF
    index ← HASH(key)
    IF NOT isSet[index] THEN
        THROW "Key not found: the
        specified key is not set."
    END IF
    isSet[index] ← false
END REMOVE
```

Symbol table - test

test_symbol_table.cpp

```cpp
#include <gtest/gtest.h>
#include "../src/symbol_table.hpp"

class SymbolTableTest : public ::testing::Test
{
protected:
    SymbolTable symbolTable;
};

// Test valid insertion and search
TEST_F(SymbolTableTest, InsertAndSearchValidKeys)
{
    symbolTable.insert('A', 1.23);
    EXPECT_DOUBLE_EQ(symbolTable.search('A'), 1.23);

    symbolTable.insert('B', 4.56);
    EXPECT_DOUBLE_EQ(symbolTable.search('B'), 4.56);

    symbolTable.insert('Z', 7.89);
    EXPECT_DOUBLE_EQ(symbolTable.search('Z'), 7.89);
}

// Test invalid key insertion
TEST_F(SymbolTableTest, InsertInvalidKey)
{
    EXPECT_THROW(symbolTable.insert('a', 1.23), std::out_of_range);
    EXPECT_THROW(symbolTable.insert('1', 4.56), std::out_of_range);
}

// Test search for unset keys
TEST_F(SymbolTableTest, SearchUnsetKey)
{
    EXPECT_THROW(symbolTable.search('C'), std::out_of_range);
    EXPECT_THROW(symbolTable.search('X'), std::out_of_range);
}

// Test remove key
TEST_F(SymbolTableTest, RemoveKey)
{
    symbolTable.insert('A', 1.23);
    symbolTable.remove('A');
    EXPECT_THROW(symbolTable.search('A'), std::out_of_range);

    symbolTable.insert('B', 4.56);
    symbolTable.remove('B');
    EXPECT_THROW(symbolTable.search('B'), std::out_of_range);
}

// Test remove invalid key
TEST_F(SymbolTableTest, RemoveInvalidKey)
{
    EXPECT_THROW(symbolTable.remove('a'), std::out_of_range);
    EXPECT_THROW(symbolTable.remove('1'), std::out_of_range);
}

// Test remove unset key
TEST_F(SymbolTableTest, RemoveUnsetKey)
{
    EXPECT_THROW(symbolTable.remove('C'), std::out_of_range);
    EXPECT_THROW(symbolTable.remove('X'), std::out_of_range);
}

// Test printST function (this would typically be done by
//capturing stdout, but it's simpler to just run it here)
TEST_F(SymbolTableTest, PrintSymbolTable)
{
    symbolTable.insert('A', 1.23);
    symbolTable.insert('B', 4.56);
    symbolTable.printST();
}

int main(int argc, char **argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Math operations

math_operations.hpp

Pseudocode

No pseudocode preview is presented because this code contains only standard C++ operations.

```cpp
#ifndef MATH_OPERATIONS_HPP
#define MATH_OPERATIONS_HPP

#include <cmath>
#include <stdexcept>
#include <algorithm>
#include <numeric>

class MathOperations
{
public:
    static double add(double a, double b) { return a + b; }
    static double subtract(double a, double b) { return a - b; }
    static double multiply(double a, double b) { return a * b; }
    static double divide(double a, double b)
    {
        if (b == 0)
            throw std::runtime_error("Division by zero");
        return a / b;
    }
    static double power(double a, double b) { return std::pow(a, b); }
    static double modulus(double a, double b) { return std::fmod(a, b); }
    static double squareRoot(double a) { return std::sqrt(a); }
    static double cubeRoot(double a) { return std::cbrt(a); }
    static double hypotenuse(double x, double y)
    {
        return std::hypot(x, y);
    }
    static double floor(double a) { return std::floor(a); }
    static double ceil(double a) { return std::ceil(a); }
    static double abs(double a) { return std::abs(a); }
    static int sign(double a) { return (a > 0) - (a < 0); }
    static double round(double a) { return std::round(a); }
    static double factorial(int n)
    {
        if (n < 0)
            throw std::invalid_argument("Factorial is not
              defined for negative numbers");
        else if (n == 1)
            return n;
        return n * factorial(n - 1);
    }
    static double log10(double a) { return std::log10(a); }
    static double log2(double a) { return std::log2(a); }
    static double perm(int n, int k)
    {
        if (n < 0 || k < 0 || k > n)
            throw std::invalid_argument("Invalid values for perm(n, k)");
        return factorial(n) / factorial(n - k);
    }
    static double comb(int n, int k)
    {
        if (n < 0 || k < 0 || k > n)
            throw std::invalid_argument("Invalid values for comb(n, k)");
        return factorial(n) / (factorial(k) * factorial(n - k));
    }
    static int gcd(int x, int y)
    {
        if (y == 0)
            return std::abs(x);
        else
            return gcd(y, x % y);
    }
    static int lcm(int x, int y)
    {
        if (x == 0 || y == 0)
            return 0;
        return std::abs(x * y) / gcd(x, y);
    }
    static double sin(double a) { return std::sin(a); }
    static double cos(double a) { return std::cos(a); }
    static double tan(double a) { return std::tan(a); }
    static double arcSin(double a) { return std::asin(a); }
    static double arcCos(double a) { return std::acos(a); }
    static double arcTan(double a) { return std::atan(a); }
};

    static bool isEqual(double a, double b) { return a == b; }
    static bool isGreater(double a, double b) { return a > b; }
    static bool isLess(double a, double b) { return a < b; }

#endif // MATH_OPERATIONS_HPP
```

Evaluator I

postfix_evaluator.hpp

```hpp
#ifndef POSTFIX_EVALUATOR_HPP
#define POSTFIX_EVALUATOR_HPP

#include <string>
#include <stdexcept>

class Result
{
public:
    bool isAssignment = false;
    bool isResult = false;
    double result;
};

#include <cmath>
#include <array>
#include <iterator>
#include <algorithm>

#include "stack.hpp"
#include "symbol_table.hpp"
#include "math_operations.hpp"

/**
 * @class PostfixEvaluator
 * @brief Evaluates postfix expressions
 * This class evaluates postfix expressions by using a stack to hold
 operands and a symbol table to hold variable values.
 */
class PostfixEvaluator
{
private:
    Stack<std::string> stack; // Stack to hold operands
    SymbolTable symbolTable;  // Symbol table to hold variable values

    /**
     * @brief Checks if a token is a number
     * This method checks if a given token is a number.
     * @param token The token to be checked
     * @return True if the token is a number, false otherwise
     */
    bool isNumber(const std::string &token) const;

    /**
     * @brief Checks if a token is a variable [A..Z]
     * This method checks if a given token is a variable [A..Z].
     * @param token The token to be checked
     * @return True if the token is a variable, false otherwise
     */
    bool isVariable(const std::string &token) const;

    /**
     * @brief Performs an operation on the stack
     * This method performs an operation on the stack using the top
     (one/two) operand(s) and the given operation.
     * @param operation The operation to be performed
     */
    void performOperation(const std::string &operation);

    // void assignKey(const std::string &key, const std::string &value)

    /**
     * @brief Evaluates the value of a token in a postfix expression.
     * This function evaluates the numeric value of a token in a
     postfix expression. If the token represents a variable, it searches
     for its value in the symbol table. If the token is a numeric literal,
     it converts it to a double and returns the value.
     * @param token A string representing a token in the postfix expression.
     * @return The numeric value of the token.
     */
    double evaluateValue(const std::string &token) const;

public:
    /**
     * @brief Evaluates a postfix expression
     * This method evaluates a postfix expression and returns the result.
     * @param expression The postfix expression to be evaluated
     * @return A Result object containing the result and whether it
     was an assignment
     */
    Result evaluate(const std::string &expression);
    void printSymbolTable();
};

#endif // POSTFIX_EVALUATOR_HPP
```

Evaluator II

postfix_evaluator.hpp | postfix_evaluator.cpp

```cpp
#include "postfix_evaluator.hpp"
#include <sstream>

double PostfixEvaluator::evaluateValue(const std::string &token) const
{
    double boolValue = (token == "true") ? 1 : (token == "false") ? 0: -1;
        if (boolValue != -1) return boolValue;

        if (isVariable(token))
        {return symbolTable.search(token[0]);}
        else
        {
            return std::stod(token);
        }
}

bool PostfixEvaluator::isNumber(const std::string &token) const
{
    try
    {
        (void)std::stod(token);
        return true;
    }
    catch (const std::invalid_argument &)
    {
        return false;
    }
}

bool PostfixEvaluator::isVariable(const std::string &token) const
{
    return (token.length() == 1 &&  token[0] >= 'A' && token[0] <= 'Z');
}


Result PostfixEvaluator::evaluate(const std::string &expression)
{
    std::istringstream iss(expression);
    std::string token;

    Result result;
    result.isAssignment = false;

    while (iss >> token)
    {
        if (isNumber(token) || isVariable(token))
        {
            stack.push(token);
        }
        else if (token == "=")
        {
            double value = evaluateValue(stack.pop());
            std::string key = stack.pop();
            if (key.length() != 1 || !isalpha(key[0]))
            {
                throw std::runtime_error("Invalid variable name");
            }
            symbolTable.insert(key[0], value);
            result.isAssignment = true;
        }
        else
        {
            performOperation(token);
        }
    }

    if (!stack.isEmpty())
    {
        result.result = evaluateValue(stack.pop());
        if (!stack.isEmpty())
        {
            stack.clearStack();
            throw std::runtime_error("Invalid expression");
        }
        result.isResult = true;
    }
    return result;
}
```

Pseudocode

```
FUNCTION EVALUATE_VALUE(token)
    IF IS_VARIABLE(token) THEN
        RETURN SYMBOL_TABLE.SEARCH(token)
    ELSE
        RETURN TO_NUMBER(token)
    END IF
END FUNCTION

FUNCTION IS_NUMBER(token)
    if token can be converted to a number:
        return TRUE
    return FALSE otherwise
END FUNCTION

FUNCTION IS_VARIABLE(token)
    IF LENGTH(token) = 1
    AND token is a letter between "A" AND "Z"
        return TRUE
    return FALSE
END FUNCTION


FUNCTION EVALUATE(expression)
    DECLARE tokens AS LIST OF STRING
      = SPLIT(expression, " ")
    DECLARE assign AS BOOL

    FOR EACH token IN tokens DO
        IF IS_NUMBER(token) OR IS_VARIABLE(token) THEN
            stack.PUSH(token)
        ELSE IF token = "=" THEN
            DECLARE value AS DOUBLE
              = EVALUATE_VALUE(stack.POP())
            DECLARE key AS STRING = stack.POP()
            IF LENGTH(key) ≠ 1 OR NOT IS_ALPHA(key) THEN
                THROW "Invalid variable name"
            END IF
            SYMBOL_TABLE.INSERT(key, value)
            assign = true
        ELSE
            PERFORM_OPERATION(token)
        END IF
    END FOR

    DECLARE  result as NULL
    IF stack.is_empty() = FALSE
        result = stack.pop()
        if stack.is_empty() = FALSE THEN
            stack.clearStack()
            THROW "Invalid expression"
    END IF
    RETURN result
END FUNCTION
```

Evaluator III

```
void PostfixEvaluator::performOperation(const std::string &operation)
{
    double a, b;

    const std::array<std::string, 27> validOperations = {
        "+",      // 0
        "-",      // 1
        "*",      // 2
        "/",      // 3
        "%",      // 4
        "pow",    // 5
        "sqrt",   // 6
        "cbrt",   // 7
        "hyp",    // 8
        "floor",  // 9
        "ceil",   // 10
        "abs",    // 11
        "sgn",    // 12
        "round",  // 13
        "!",      // 14
        "log10",  // 15
        "log2",   // 16
        "perm",   // 17
        "comb",   // 18
        "gcd",    // 19
        "lcm",    // 20
        "sin",    // 21
        "cos",    // 22
        "tan",    // 23
        "asin",   // 24
        "acos",   // 25
        "atan"    // 26
        "ge",     // 27
        "le",     // 28
        "=="      // 29
    };
```

```
FUNCTION PERFORM_OPERATION(operation)
    DECLARE a, b AS DOUBLE
    IF operation = "+" THEN
        b ← EVALUATE_VALUE(stack.POP())
        a ← EVALUATE_VALUE(stack.POP())
        stack.PUSH(
          TO_STRING(MathOperations.ADD(a, b))
        )
    ...
    ELSE
      THROW "Invalid token in expression"
    END IF
END FUNCTION
```

```
auto it = std::find(validOperations.begin(),
          validOperations.end(), operation);
    if (it == validOperations.end())
    {
        throw std::runtime_error("Invalid token in expression: "
          + operation);
    }

    switch (std::distance(validOperations.begin(), it))
    {
    case 0: // "+"
        b = evaluateValue(stack.pop());
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::add(a, b)));
        break;
    case 1: // "-"
        b = evaluateValue(stack.pop());
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::subtract(a, b)));
        break;
    case 2: // "*"
        b = evaluateValue(stack.pop());
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::multiply(a, b)));
        break;
    case 3: // "/"
        b = evaluateValue(stack.pop());
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::divide(a, b)));
        break;
    case 4: // "%"
        b = static_cast<int>(evaluateValue(stack.pop()));
        a = static_cast<int>(evaluateValue(stack.pop()));
        stack.push(std::to_string(MathOperations::modulus(a, b)));
        break;
    case 5: // "pow"
        b = evaluateValue(stack.pop());
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::power(a, b)));
        break;
    case 6: // "sqrt"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::squareRoot(a)));
        break;
```

Evaluator IV

```
    case 7: // "cbrt"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::cubeRoot(a)));
        break;
    case 8: // "hyp"
        b = evaluateValue(stack.pop());
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::hypotenuse(a, b)));
    case 9: // "floor"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::floor(a)));
        break;
    case 10: // "ceil"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::ceil(a)));
        break;
    case 11: // "abs"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::abs(a)));
        break;
    case 12: // "sgn"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::sign(a)));
        break;
    case 13: // "round"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::round(a)));
        break;
    case 14: //"!"
        a = static_cast<int>(evaluateValue(stack.pop()));
        stack.push(std::to_string(MathOperations::factorial(a)));
        break;
    case 15: // "log10"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::log10(a)));
        break;
    case 16: // "log2"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::log2(a)));
        break;
    case 17: // "perm"
        b = static_cast<int>(evaluateValue(stack.pop()));
        a = static_cast<int>(evaluateValue(stack.pop()));
        stack.push(std::to_string(MathOperations::perm(a, b)));
        break;
    case 18: // "comb"
        b = static_cast<int>(evaluateValue(stack.pop()));
        a = static_cast<int>(evaluateValue(stack.pop()));
        stack.push(std::to_string(MathOperations::comb(a, b)));
        break;
    case 19: // "gcd"
        b = static_cast<int>(evaluateValue(stack.pop()));
        a = static_cast<int>(evaluateValue(stack.pop()));
        stack.push(std::to_string(MathOperations::gcd(a, b)));
        break;
    case 20: // "lcm"
        b = static_cast<int>(evaluateValue(stack.pop()));
        a = static_cast<int>(evaluateValue(stack.pop()));
        stack.push(std::to_string(MathOperations::lcm(a, b)));
        break;
    case 21: // "sin"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::sin(a)));
        break;
    case 22: // "cos"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::cos(a)));
        break;
    case 23: // "tan"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::tan(a)));
        break;
    case 24: // "asin"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::arcSin(a)));
        break;
    case 25: // "acos"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::arcCos(a)));
        break;
    case 26: // "atan"
        a = evaluateValue(stack.pop());
        stack.push(std::to_string(MathOperations::arcTan(a)));
        break;
    }
}
```

• • •

Evaluator - test

h test_postfix_operator.cpp

```cpp
#include "../src/postfix_evaluator.hpp"
#include <gtest/gtest.h>

class PostfixEvaluatorTest : public testing::Test
{
protected:
    void SetUp() override
    {
        evaluator = new PostfixEvaluator();
    }

    void TearDown() override
    {
        delete evaluator;
    }

    PostfixEvaluator *evaluator;
};

// Test cases for evaluate function
TEST_F(PostfixEvaluatorTest, Evaluate_Assignment)
{
    // Test postfix expression "A 3 ="
    Result result = evaluator->evaluate("A 3 =");

    // Verify that it's an assignment
    EXPECT_TRUE(result.isAssignment);

    // Verify that the symbol table contains the assignment
    Result result = evaluator->evaluate("A");
    EXPECT_DOUBLE_EQ(result.result, 3.0);
}

TEST_F(PostfixEvaluatorTest, Evaluate_NoAssignment)
{
    // Test postfix expression "10 20 +"
    Result result = evaluator->evaluate("10 20 +");

    // Verify that it's not an assignment
    EXPECT_FALSE(result.isAssignment);

    // Verify the result of the expression
    EXPECT_DOUBLE_EQ(result.result, 30.0);
}

int main(int argc, char **argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Main I

main.cpp

```cpp
#include <iostream>
#include <string>
#include "postfix_evaluator.hpp"

// ANSI color codes
const std::string RESET = "\033[0m";
const std::string RED = "\033[31m";
const std::string CYAN = "\033[36m";
const std::string WHITE = "\033[37m";
const std::string SOFT_PINK = "\033[38;5;225m";

/**
 * @brief Display the allowed variables and usage instructions.
 */
void displayInstructions()
{
    std::cout << SOFT_PINK << "\nPostfix Expression Evaluator\n";
    std::cout << "============================\n";
    std::cout << "Allowed variables: [A-Z]\n";
    std::cout << "Operations supported:\n";
    std::cout << " +   : Addition\n";
    std::cout << " -   : Subtraction\n";
    std::cout << " *   : Multiplication\n";
    std::cout << " /   : Division\n";
    std::cout << " %   : Modulus\n";
    std::cout << " floor : Floor function\n";
    std::cout << " ceil  : Ceiling function\n";
    std::cout << " round : Round function\n";
    std::cout << " abs   : Absolute value\n";
    std::cout << " cos   : Cosine\n";
    std::cout << " sin   : Sine\n";
    std::cout << " tan   : Tangent\n";
    std::cout << " log10 : Logarithm base 10\n";
    std::cout << " log2  : Logarithm base 2\n";
    std::cout << " pow   : Power function\n";
    std::cout << " sqrt  : Square root function\n";
    std::cout << " cbrt  : Cube root function\n";
    std::cout << " hyp   : Hypotenuse function\n";
    std::cout << " asin  : Arcsine function\n";
    std::cout << " acos  : Arccosine function\n";
    std::cout << " atan  : Arctangent function\n";
    std::cout << " !     : Factorial function\n";
    std::cout << " sgn   : Sign function\n";
    std::cout << " perm  : Permutations function\n";
    std::cout << " comb  : Combinations function\n";
    std::cout << " gcd   : Greatest Common Divisor function\n";
    std::cout << " lcm   : Least Common Multiple function\n";
    std::cout << " ==    : Equality comparison (TRUE = 1, FALSE = 0)\n";
    std::cout << " >     : Greater than comparison (TRUE = 1, FALSE = 0)\n";
    std::cout << " <     : Less than comparison (TRUE = 1, FALSE = 0)\n";
    std::cout << "Enter 'exit' to quit the program.\n";
    std::cout << "Note: Ensure your expressions are space-separated.\n";
    std::cout << "Example: A 5 = B 3 = A B +\n";
    std::cout << "============================\n";
    std::cout << "Write 'vars' to display assigned variables.\n"
              << RESET;
}
```

Main II

```cpp
/**
 * @brief The main function of the program.
 *
 * This function handles user input, evaluates the postfix expression
 using the PostfixEvaluator, and prints the result to the terminal.
 */
int main()
{
    displayInstructions();
    PostfixEvaluator evaluator;
    std::string input;

    while (true)
    {
        std::cout << "\n"
                  << CYAN << "Enter postfix expression: "
                             "\n"
                  << RESET;
        std::getline(std::cin, input);

        if (input == "exit")
        {
            break;
        }
        else if (input == "vars")
        {
            evaluator.printSymbolTable();
        }
        else
        {
            try
            {
                Result result = evaluator.evaluate(input);
                if (result.isAssignment == true)
                    std::cout << "Assignment successful.\n";
                if (result.isResult == true)
                    std::cout << SOFT_PINK << "Result: " << result.result << "\n"
                              << RESET;
            }
            catch (const std::exception &e)
            {
                std::cerr << RED << "Error: " << e.what() << "\n"
                          << RESET;
            }
        }
    }

    std::cout << SOFT_PINK << "Exiting program. Goodbye!\n"
              << RESET;
    return 0;
}
```

Postfix++ video

New Tab

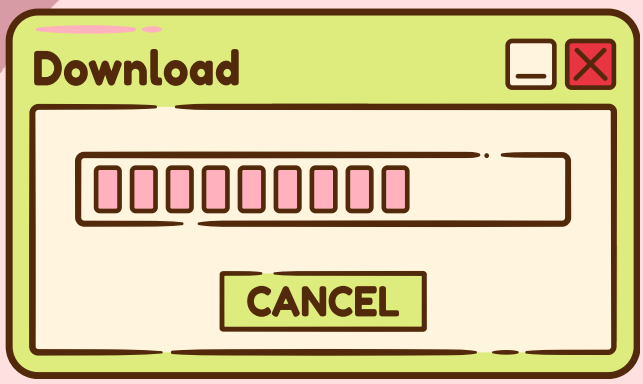
Postfix++ ADS II Mid-term

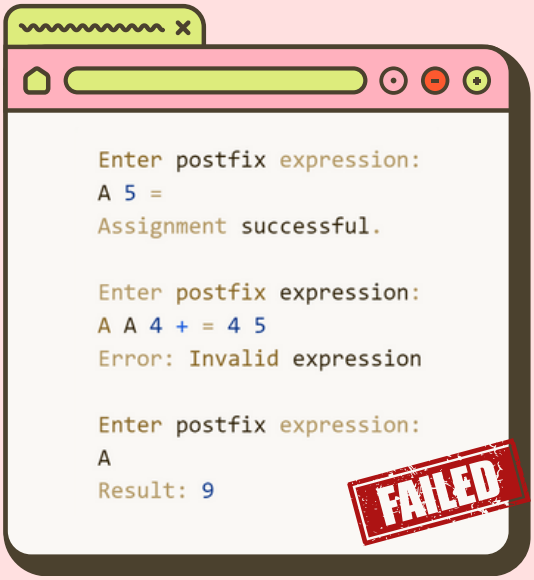
Vimeo

CLICK HERE

Section 6.

Defects and Improvements





A notable deficiency in the current implementation is the lack of clear and descriptive error messages, which makes it difficult to debug incorrect expressions. For example, when an invalid expression such as "A A 4 + = 4 5" is entered after assigning a value to "A", the program simply displays a generic "Error: Invalid expression" message without providing details about the reason for the error. Additionally, another related issue was identified with the improper handling of the stack after certain operations. For instance, after entering "A 5 =" and then "A A 4 + = 4 5", the value 9 is assigned to A, and upon simply entering "A", the value 9 is displayed instead of the value previously assigned to "A" (5). This occurs because the current implementation does not perform anticipatory validation of the entered expression before proceeding to evaluate it and make changes to the stack or hash table.

To address these deficiencies, it is suggested to implement an expression validation logic that verifies the correctness of the entered expression before proceeding to evaluate it. This validation could iterate through the expression string and check if it contains valid operators and if each operator has the correct number of operands. Additionally, if an operand is a hash table key, the validation should check if that key has been previously defined. If any errors are detected during the validation, a descriptive and specific error message should be displayed, indicating the problem found, instead of a generic message. This solution would improve the user experience by providing more informative feedback and prevent incorrect operations from being performed on the stack or hash table when invalid expressions are entered.

RESOURCES

[1] cplusplus.com. Arrays.

[2] Canva - creating the report visuals.

[3] CodeToImg - converting code snippets into images.


THANKS!
"Thank C:"
.—B