Algorithms and Data Structures II
Mid-terms Coursework Report
(Postfix++)

Name: TAN YEE CHONG (CHEN YICONG)

Student Number (S/N): 230668566

Module Number: CM2035

Submission Date: 8 July 2024

Table of Contents

Section 1: My Solution

For this project task, I have implemented a C++ interpreter for Postfix. I utilized Stack and Hashtable data structures to create an interpreter capable of variable storage and arithmetic operations based on the array data structure. The Stack is used for storing operands and results, enabling sequential execution of arithmetic operations. This data structure is well-suited for the task, as its Last-In-First-Out (LIFO) nature allows for storing and retrieving operands in the exact order needed for postfix calculations. This ensures efficient and straightforward computation, as each operation can directly use the most recently processed values. The Hashtable serves as a symbol table for variable storage. I have customized the Hashtable to handle a limited set of keys (A-Z) due to memory limitations as reflected in the requirements. This simplifies the implementation and is suitable for the coursework's scope and the targeted hardware, which is a mobile device with very limited memory. The Hashtable offers fast access to stored values, allowing for quick retrieval and updating of variable data, which is crucial for dynamic calculations. I have used a class-based approach for the design, separating the concerns of arithmetic operations and symbol table management by creating two classes: Interpreter and Hashtable. This modular design helps maintain the code's organization and flexibility. Additionally, I have added some error handling in my version of Postfix++. My code also includes functions to delete nodes, search, and insert, as simulated by the requirements.

Section 2: Explanation of algorithms

I used a stack to do mathematical operations, popping operands that had been inserted before, processing them, and then pushing the outcome back onto the stack.
Every operation in my algorithm pulls its parameters from the stack when it functions properly. The number of inputs can differ; for example, the "abs" function only needs one value to be popped, whereas the "+" action requires two. After doing the calculations, the operation returns the outcome to the stack.

To ensure user input is correct, the state of the stack is being checked. After the user input has been processed, the stack must be either empty (in case of assignment operation) or hold a single value which is used in an implicit “return” operation. If the interpreter “returns” a value, it is then printed back to the user.

Using “B 12 =” as example, the algorithm breakdown of the user’s input as follows:

Input	Operation	Stack	Symbol Table
B	Push B	[B]	{}
12	Push 12	[B, 12]	{}
=	Assignment	[]	{ "B": 12 }

The assignment op pops 12 as a value, then pops “B” as a variable name, and saves the variable in the symbol table. The stack remains empty.

Figure 1: Image Demonstration of B 12 =

Consider the simple example of the postfix operation "12 12 *". Here's a breakdown of the algorithm's execution:

Step	Operation	Stack	Time Comp	Space Comp
1	Push 12 onto Stack	[12]	O(1)	O(1)
2	Push 12 onto Stack	[12, 12]	O(1)	O(1)
3	Perform Operation	[]	O(1)	O(1)
4	Push 144 onto Stack	[144]	O(1)	O(1)

As seen in the table, each step involves constant time complexity as individual elements are pushed, popped, or operated upon within the stack. The space complexity remains constant throughout the execution, as the size of the stack does not vary with input size or number of operations.

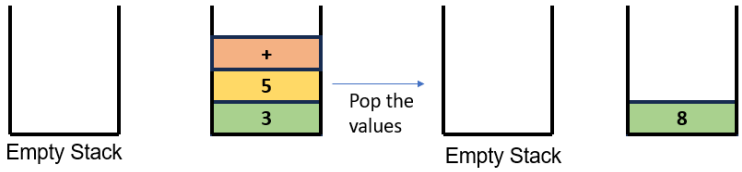

Figure 2. Visual Demonstration how the interpreter works in a stack

Symbol Table

The easiest way to implement a symbol table would be using a dictionary (associative array) data structure. The task description explicitly disallows using any data structure other than an Array, so the only way is to implement a simple HashTable-based dictionary is using an array as storage.

Given the limitation to possible variable names, we can make the implementation simpler since there are only 26 possible names (“A” to “Z”). This means a hash table with an initial 26 buckets will never have to expand, and neither will it have any collisions.

Since there is no possible collisions, the best method to is to use HashTable-based dictionary using as an Array as the time complexity of insert and search is *O(1)*, where 1, or *n*, represent the number of elements the algorithm needs to iterate before finding the key and it is more memory efficient.

Collision Resolution	Data Structure	Insertion (Average Case)	Insertion (Worst Case)	Search (Average Case)	Search (Worst Case)
Separate Chaining	Array	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Separate Chaining	Linked List	$O(1)$	$O(n)$	$O(1)$	$O(n)$

Figure 3. The Average Case and Worst Case of Time complexity of Hash Table

Now, consider the following operation and variable assignment.

Input > **B** 200 =

Input > end

postfix operation:

Input > **B** log10

Output > 2.301

Line of Code	Action	Time Comp. $O(?)$	Stack	Symbol Table
`tokenize("B log10")`	Tokenizing input	$O(n)$	`NULL`	`{B: 200}`
`run(tokens)`	Iterating over tokens	$O(n)$	`NULL`	`{B: 200}`
`resolve('B')`	Get variable 'B'	$O(1)$	`200`	`{B: 200}`
`hashFunc('B')`	Hash function calculation	$O(1)$	`200`	`{B: 200}`
`search('B')`	Retrieve from hash table	$O(1)$	`200`	`{B: 200}`
`push(200)`	Pushing onto stack	$O(1)$	`[200]`	`{B: 200}`
`log10()`	`log10` operation	$O(1)$	`[2.301]`	`{B: 200}`

Figure 4. An Overview of the application step-by-step execution for B log10

The hash function calculates the index for the variable "B" efficiently, guaranteeing constant-time complexity access to the matching bucket in the hash table. Likewise, retrieving the value associated with "B" also involves constant-time complexity, facilitating rapid variable lookup.

As per the requirements, the interpreter implements a straightforward validation mechanism to limit variable names to the range from "A" to "Z". Any attempt to use an invalid variable name triggers the detection of incorrect input, leading to the raising of an error that indicates the variable should be within the range of A to Z.

The Symbol Table, also known as HashTable, can be represented as a box with index-labelled compartments.

The indexing is done by using the hash function which simply subtracts the ASCII value of 'A' from the ASCII value of the input character key. This effectively maps the capital letters 'A' to 'Z' (ASCII values 65 to 90) to the range of indices 0 to 25, which corresponds to the size of the values array.

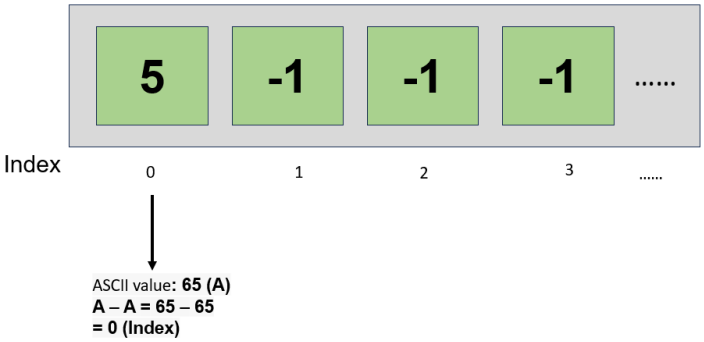

Figure 5

In figure 5, if the input character is 'A', the hash function will return 0 (since 'A' - 'A' = 0). If the input character is 'Z', the hash function will return 25 (since 'Z' - 'A' = 25). This simple hash function is used to directly map the characters 'A' to 'Z' to the corresponding indices in the values array. Note that indexes with -1 are empty “basket” with no value assigned to it.

The load factor of my Hash-table would be having a maximum load factor of 1.0, if all the variable ‘A’ to ‘Z’ is filled up.

Section 3: Pseudocode

Hashtable.cpp

```
HASH_TABLE_INITIALIZE()
    for i ← 0 to LETTERS_COUNT - 1
        VALUES[i] ← NaN

HASH_FUNC(key)
    return key - 'A'

CHECK_VARIABLE(key)
    if key < 'A' or key > 'Z'
        ERROR("Variables should be from A to Z")

HASH_TABLE_INSERT(key, value)
    CHECK_VARIABLE(key)
    index ← HASH_FUNC(key)
    VALUES[index] ← value

HASH_TABLE_SEARCH(key)
    CHECK_VARIABLE(key)
    index ← HASH_FUNC(key)
    if VALUES[index] = NaN
        ERROR("Unknown variable " + key)
    return VALUES[index]

HASH_TABLE_REMOVE(key)
    CHECK_VARIABLE(key)
    index ← HASH_FUNC(key)
    VALUES[index] ← NaN

HASH_TABLE_PRINT()
    for i ← 0 to LETTERS_COUNT - 1
        variable ← 'A' + i
        if VALUES[i] = NaN
            PRINT(variable + " = N/A")
        else
            PRINT(variable + " = " + VALUES[i])

HASH_TABLE_CLEAR()
    for i ← 0 to LETTERS_COUNT - 1
        VALUES[i] ← NaN
```

Interpreter.cpp

```
INTERPRETER_INITIALIZE()
    stackTop ← -1

STACK_POP()
    if stackTop < 0
        ERROR("stack is empty")
    value ← stack[stackTop]
    stackTop ← stackTop - 1
    return value

PEEK()
```

```
    if stackTop < 0
        ERROR("stack is empty")
    return stack[stackTop]

RESOLVE(value)
    return value

RESOLVE(variable)
    return SYMBOL_TABLE_SEARCH(variable)

ADD()
    right ← RESOLVE(STACK_POP())
    left ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← left + right

MULTIPLY()
    right ← RESOLVE(STACK_POP())
    left ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← left * right

SUBTRACT()
    right ← RESOLVE(STACK_POP())
    left ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← left - right

DIVIDE()
    right ← RESOLVE(STACK_POP())
    left ← RESOLVE(STACK_POP())
    if right = 0
        ERROR("division by zero")
    stackTop ← stackTop + 1
    stack[stackTop] ← left / right

MODULO()
    right ← RESOLVE(STACK_POP())
    left ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← left mod right

ROUND()
    value ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← ROUND(value)

FLOOR()
    value ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← FLOOR(value)

CEIL()
    value ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← CEIL(value)

ABS()
```

```
    value ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← ABS(value)

COS()
    value ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← COS(value)

SIN()
    value ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← SIN(value)

TAN()
    value ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← TAN(value)

LOG10()
    value ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← LOG10(value)

LOG2()
    value ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← LOG2(value)

EXP()
    value ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← EXP(value)

FACTORIAL()
    value ← RESOLVE(STACK_POP())
    stackTop ← stackTop + 1
    stack[stackTop] ← CALCULATE_FACTORIAL(value)

COT()
    value ← RESOLVE(STACK_POP())
    sinValue ← SIN(value)
    if sinValue = 0
        ERROR("Cotangent is undefined for sin(x) = 0")
    stackTop ← stackTop + 1
    stack[stackTop] ← COS(value) / sinValue

CALCULATE_FACTORIAL(n)
    if n = 0 or n = 1
        return 1
    else
        return n * CALCULATE_FACTORIAL(n - 1)

MAX()
    if stackTop < 1
        ERROR("Not enough elements in the stack to perform max
operation.")
    right ← RESOLVE(STACK_POP())
```

```
    left ← RESOLVE(STACK_POP())
    maximum ← MAX(left, right)
    stackTop ← stackTop + 1
    stack[stackTop] ← maximum

MIN()
    if stackTop < 1
        ERROR("Not enough elements in the stack to perform min
operation.")
    right ← RESOLVE(STACK_POP())
    left ← RESOLVE(STACK_POP())
    minimum ← MIN(left, right)
    stackTop ← stackTop + 1
    stack[stackTop] ← minimum

RUN(tokens, tokenCount)
    for i ← 0 to tokenCount - 1
        token ← tokens[i]

        if IS_DIGIT(token[0]) or (token[0] = '-' and LENGTH(token) >
1)
            stackTop ← stackTop + 1
            stack[stackTop] ← CONVERT_TO_DOUBLE(token)
        else if LENGTH(token) = 1 and IS_ALPHA(token[0])
            stackTop ← stackTop + 1
            stack[stackTop] ← RESOLVE(token[0])
        else
            if token = "+"
                ADD()
            else if token = "*"
                MULTIPLY()
            else if token = "-"
                SUBTRACT()
            else if token = "/"
                DIVIDE()
            else if token = "%"
                MODULO()
            else if token = "round"
                ROUND()
            else if token = "floor"
                FLOOR()
            else if token = "ceil"
                CEIL()
            else if token = "abs"
                ABS()
            else if token = "cos"
                COS()
            else if token = "sin"
                SIN()
            else if token = "tan"
                TAN()
            else if token = "log10"
                LOG10()
            else if token = "log2"
                LOG2()
            else if token = "exp"
                EXP()
            else if token = "!"
```

```
                FACTORIAL()
            else if token = "cot"
                COT()
            else if token = "max"
                MAX()
            else if token = "min"
                MIN()
            else
                ERROR("Unknown operator: " + token)
    if stackTop ≠ 0
        ERROR("stack length is " + (stackTop + 1))

GET_RESULT()
    if stackTop < 0
        ERROR("stack is empty")
    return stack[stackTop]

ASSIGN_VARIABLE(varName, varValue)
    SYMBOL_TABLE_INSERT(varName, varValue)
```

main.cpp

```
CONSTANT MAX_TOKENS ← 100

FUNCTION USER_INPUT(line, tokens, tokenCount)
    INITIALIZE ss AS STRING_STREAM(line)
    INITIALIZE tokenCount AS 0
    WHILE ss CAN EXTRACT token AND tokenCount < MAX_TOKENS
        tokens[tokenCount] ← token
        tokenCount ← tokenCount + 1

FUNCTION MAIN()
    INITIALIZE interpreter AS NEW INTERPRETER
    INITIALIZE line AS EMPTY STRING
    INITIALIZE tokens AS ARRAY OF MAX_TOKENS STRINGS
    INITIALIZE tokenCount AS 0

    PRINT "Enter variable assignments (e.g., A 3 =). Type 'clear' to
reset hashtable or 'end' to finish:"

    WHILE TRUE
        PRINT "> "
        READ line FROM USER INPUT
        IF line = "end"
            BREAK

        IF line = "clear"
            CALL interpreter.symbolTable.CLEAR()
            PRINT "HashTable cleared."
            CONTINUE

        USER_INPUT(line, tokens, tokenCount)

        IF tokenCount = 3 AND tokens[2] = "="
            varName ← FIRST CHARACTER OF tokens[0]
            varValue ← CONVERT tokens[1] TO DOUBLE
```

```
            CALL interpreter.ASSIGN_VARIABLE(varName, varValue)
            CALL interpreter.symbolTable.PRINT()
        ELSE
            PRINT "Invalid input format for variable assignment."

    PRINT "Enter the postfix expression:"
    PRINT "> "
    READ line FROM USER INPUT
    USER_INPUT(line, tokens, tokenCount)

    TRY
        CALL interpreter.RUN(tokens, tokenCount)
        PRINT "The result is: " + CALL interpreter.GET_RESULT()
    CATCH EXCEPTION e
        PRINT ERROR e.MESSAGE()

    RETURN 0

CALL MAIN()
```

Section 4: Data Structure

The choice of data structures for the Postfix++ interpreter aligns with the task requirements. The use of a Stack for arithmetic operations and a HashTable for variable storage, both based on a built-in Array, is well-suited for the implementation.

The Stack data structure is ideal for sequential arithmetic operations due to its **Last-In-First-Out (LIFO) nature**. It allows for storing and retrieving operands in the exact order needed for postfix calculations. This ensures efficient and straightforward computation, as each operation can directly use the most recently processed values.

The HashTable, on the other hand, is chosen for variable storage because it offers fast access to stored values. By mapping variable names (keys) to their corresponding values, the HashTable enables quick retrieval and updating of variable data, which is crucial for dynamic calculations. The customization to handle a limited range of keys (A-Z) simplifies the implementation and is suitable for the coursework's scope and the targeted hardware, which is a mobile device with very limited memory.

Hash-Table (using an array) vs other methods of implementation

For the choices of implementation of the symbol table, we shall look at the time complexity for search and insertion, flexibility in the key types.

- Time complexity

    - Binary search trees have the time complexity $O(logn)$ for insertion and traversing though the data.
    - Hashtable as an array *has* average-case $O(1)$ time complexity for lookups, insertions, and deletions

Hence, choosing an HashTable is a better choice in terms of Time complexity.

- Flexibility in the key types

    - Binary search trees can handle various key types, but with added complexity.
    - Hashtable can handle a variety of key types (e.g., strings, integers), making them versatile for storing variable names.

Hence, choosing an HashTable is a better choice in terms of Key Types Flexibility.

Section 5: Full Source Code and Video Demonstration

Appendix 1. Link to the video demonstration

Here's the link to the video demonstration:

https://youtu.be/LnhFPm_ygKI

Appendix 2. Source code (C++)

HashTable.cpp

```
#include "HashTable.h"
#include <cmath>
#include <stdexcept>
#include <limits>
// Constructor: initialize the values array with NaN
HashTable::HashTable() {
    for (int i = 0; i < LETTERS_COUNT; ++i) {
        values[i] = std::numeric_limits<double>::quiet_NaN();
    }
}
// Hash function to convert a character to an index
int HashTable::hashFunc(char key) {
    return key - 'A';
}
// Function to check if the character is a valid variable (A-Z)
void HashTable::checkVariable(char key) {
    if (key < 'A' || key > 'Z') {
        throw std::runtime_error("Variables should be from A to Z");
    }
}
// Insert a value into the hash table
void HashTable::insert(char key, double value) {
    checkVariable(key);
    int index = hashFunc(key);
    values[index] = value;
}
// Search for a value in the hash table
double HashTable::search(char key) {
    checkVariable(key);
    int index = hashFunc(key);
    if (std::isnan(values[index])) {
        throw std::runtime_error("Unknown variable \"" +
std::string(1, key) + "\"");
    }
    return values[index];
}
// Remove a value from the hash table
void HashTable::remove(char key) {
    checkVariable(key);
    int index = hashFunc(key);
    values[index] = std::numeric_limits<double>::quiet_NaN();
}

void HashTable::clear()
{
```

```
     for (int i = 0; i < LETTERS_COUNT; i++) {
        values[i] = std::numeric_limits<double>::quiet_NaN();
    }
}
```

Hashtable.h

```
class HashTable {
private:
    static const int LETTERS_COUNT = 26;
    double values[LETTERS_COUNT];
    int hashFunc(char key);
    void checkVariable(char key);
public:
    HashTable();
    void insert(char key, double value);
    double search(char key);
    void remove(char key);
    void clear();
};
```

Interpreter.cpp

```
#include "Interpreter.h"
#include <cmath>
#include <stdexcept>
#include <cctype>
Interpreter::Interpreter() : stackTop(-1) {}
double Interpreter::stackPop() {
    if (stackTop < 0) {
        throw std::runtime_error("stack is empty");
    }
    return stack[stackTop--];
}

double Interpreter::peek() const {
    if (stackTop < 0) {
        throw std::runtime_error("stack is empty");
    }
    return stack[stackTop];
}
double Interpreter::resolve(double value) {
    return value;
}
double Interpreter::resolve(char var) {
    return symbolTable.search(var);
}
void Interpreter::add() {
    double right = resolve(stackPop());
    double left = resolve(stackPop());
    stack[++stackTop] = left + right;
}
void Interpreter::multiply() {
    double right = resolve(stackPop());
    double left = resolve(stackPop());
    stack[++stackTop] = left * right;
}
void Interpreter::subtract() {
```

```
    double right = resolve(stackPop());
    double left = resolve(stackPop());
    stack[++stackTop] = left - right;
}
void Interpreter::divide() {
    double right = resolve(stackPop());
    double left = resolve(stackPop());
    if (right == 0) {
        throw std::runtime_error("division by zero");
    }
    stack[++stackTop] = left / right;
}
void Interpreter::modulo() {
    double right = resolve(stackPop());
    double left = resolve(stackPop());
    stack[++stackTop] = fmod(left, right);
}
void Interpreter::round() {
    double value = resolve(stackPop());
    stack[++stackTop] = ::round(value);
}
void Interpreter::floor() {
    double value = resolve(stackPop());
    stack[++stackTop] = ::floor(value);
}
void Interpreter::ceil() {
    double value = resolve(stackPop());
    stack[++stackTop] = ::ceil(value);
}
void Interpreter::abs() {
    double value = resolve(stackPop());
    stack[++stackTop] = ::fabs(value);
}
void Interpreter::cos() {
    double value = resolve(stackPop());
    stack[++stackTop] = ::cos(value);
}
void Interpreter::sin() {
    double value = resolve(stackPop());
    stack[++stackTop] = ::sin(value);
}
void Interpreter::tan() {
    double value = resolve(stackPop());
    stack[++stackTop] = ::tan(value);
}
void Interpreter::log10() {
    double value = resolve(stackPop());
    stack[++stackTop] = ::log10(value);
}
void Interpreter::log2() {
    double value = resolve(stackPop());
    stack[++stackTop] = ::log2(value);
}
void Interpreter::exp() {
    double value = resolve(stackPop());
    stack[++stackTop] = ::exp(value);
}
void Interpreter::factorial() {
```

```
    double value = resolve(stackPop());
    stack[++stackTop] = calculateFactorial(value);
}
void Interpreter::cot() {
    double value = resolve(stackPop());
    double sinValue = std::sin(value);
    if (sinValue == 0) {
        throw std::runtime_error("Cotangent is undefined for sin(x) =
0");
    }
    stack[++stackTop] = std::cos(value) / sinValue;
}
double Interpreter::calculateFactorial(double n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * calculateFactorial(n - 1);
    }
}
void Interpreter::max() {
    if (stackTop < 1) {
        throw std::runtime_error("Error: Not enough elements in the
stack to perform max operation.");
    }
    double right = resolve(stackPop());
    double left = resolve(stackPop());
    double maximum = std::max(left, right);
    stack[++stackTop] = maximum;
}
void Interpreter::min() {
    if (stackTop < 1) {
        throw std::runtime_error("Error: Not enough elements in the
stack to perform min operation.");
    }
    double right = resolve(stackPop());
    double left = resolve(stackPop());
    double minimum = std::min(left, right);
    stack[++stackTop] = minimum;
}
void Interpreter::run(const std::string tokens[], int tokenCount) {
    for (int i = 0; i < tokenCount; ++i) {
        const std::string& token = tokens[i];
        if (isdigit(token[0]) || (token[0] == '-' && token.size() >
1)) {
            stack[++stackTop] = std::stod(token);
        } else if (token.size() == 1 && isalpha(token[0])) {
            stack[++stackTop] = resolve(token[0]);
        } else {
            if (token == "+") add();
            else if (token == "*") multiply();
            else if (token == "-") subtract();
            else if (token == "/") divide();
            else if (token == "%") modulo();
            else if (token == "round") round();
            else if (token == "floor") floor();
            else if (token == "ceil") ceil();
            else if (token == "abs") abs();
            else if (token == "cos") cos();
```

```
            else if (token == "sin") sin();
            else if (token == "tan") tan();
            else if (token == "log10") log10();
            else if (token == "log2") log2();
            else if (token == "exp") exp();
            else if (token == "!") factorial();
            else if (token == "cot") cot();
            else if (token == "max") max();
            else if (token == "min") min();
            else {
                throw std::runtime_error("Unknown operator: " +
token);
            }
        }
    }
    if (stackTop != 0) {
        throw std::runtime_error("stack length is " +
std::to_string(stackTop + 1));
    }
}
double Interpreter::getResult() {
    if (stackTop < 0) {
        throw std::runtime_error("stack is empty");
    }
    return stack[stackTop];
}
void Interpreter::assignVariable(char varName, double varValue) {
    symbolTable.insert(varName, varValue);
}
```

Interpreter.h

```
#include "HashTable.h"
#include <string>
class Interpreter {
private:
    static const int STACK_MAX_SIZE = 100;
    double stack[STACK_MAX_SIZE];
    int stackTop;
    HashTable symbolTable;
    double stackPop();
    double resolve(double value);
    double resolve(char var);
    void add();
    void multiply();
    void subtract();
    void divide();
    void modulo();
    void round();
    void floor();
    void ceil();
    void abs();
    void cos();
    void sin();
    void tan();
    void log10();
```

```
    void log2();
    void exp();
    void factorial();
    void cot();
    void max();
    void min();
    double calculateFactorial(double n);
public:
    Interpreter();
    void run(const std::string tokens[], int tokenCount);
    double getResult();
    double peek() const;
    void assignVariable(char varName, double varValue);
};
```

main.cpp

```
#include <iostream>
#include <string>
#include <sstream>
#include <stdexcept>
#include "Interpreter.h"
using namespace std;
const int MAX_TOKENS = 100; // Define a maximum number of tokens
// Convert user input line to tokens and store them in a fixed-size
array
void userInput(const string& line, string tokens[], int& tokenCount)
{
    stringstream ss(line);
    string token;
    tokenCount = 0;
    while (ss >> token && tokenCount < MAX_TOKENS) {
        tokens[tokenCount++] = token;
    }
}
int main() {
    Interpreter interpreter;
    string line;
    string tokens[MAX_TOKENS];
    int tokenCount;
    cout << "Enter variable assignments (e.g., A 3 =). Type 'clear'
to reset hashtable or 'end' to finish:" << endl;
    while (true) {
        cout << "> ";
        getline(cin, line);
        if (line == "end") break;
        if (line == "clear") {
            interpreter.symbolTable.clear();
            cout << "HashTable cleared." << endl;
            continue; // Skip the rest of the loop and go to the next
iteration
        }
        userInput(line, tokens, tokenCount);
        if (tokenCount == 3 && tokens[2] == "=") {
            char varName = tokens[0][0];
            double varValue = stod(tokens[1]);
            interpreter.assignVariable(varName, varValue);
            interpreter.symbolTable.print();
```

```
        } else {
            cout << "Invalid input format for variable assignment."
<< endl;
        }
    }
    cout << "Enter the postfix expression:" << endl;
    cout << "> ";
    getline(cin, line);
    userInput(line, tokens, tokenCount);
    try {
        interpreter.run(tokens, tokenCount);
        cout << "The result is: " << interpreter.getResult() << endl;
    } catch (const exception& e) {
        cerr << e.what() << endl;
    }
    return 0;
}
```

Section 6: Defects and Improvements

My version of the PostFix++ application has the following defects:

- This implementation has a limited variable range. The hash table only handles a restricted set of keys as constrained by the requirements.
- The Stack used in the program has a fixed size of 100 elements.
- The array-based implantation of the Hash Table may have a poor performance if there's a high rate of collusions.
- The programme may not thoroughly validate user input, introducing run-time errors or unexpected errors.

My proposal to address these shortcomings:

- Expand the variables range. Implement a more versatile hash function to handle a broader range of keys.
  (a) Expanding the variables range bound to meet with collusions with indexing data (key). Hence, I suggest using **separate chaining** to resolve any collusions. Using Figure 3, separate chaining deals with collusion the best with the least time complexity.

Hashing	Separate chaining* (array of lists)	O(1)	O(1)	O(1)	O(1)	O(n+m)
	Linear probing* (parallel arrays)	O(lg(n))	O(lg(n))	O(1)	O(1)	O(n)

- with uniform and independent hash function
- n : Number of the inserted elements
- m : Size of Array of the list (The range of the hashing function results)

Figure 6

  (b) Utilize dynamic array resizing. To optimise memory usage, allow the hash table to resize based on the number of stored elements.

- Implement the stack using a dynamic array (e.g., std::vector in C++ or ArrayList in Java) which can grow or shrink as needed.

- Use Linked-list instead of Array as seen in Figure 7

Access Time	O(n) for collision resolution	O(1) if no collision, worse with clustering
Handling High Load Factors	Better performance with high collision rates	Performance degrades with high load factors

Figure 7

- Implement more comprehensive error handling mechanisms to address invalid input scenarios. This will allow the system to manage errors and provide users with clear guidance on how to use the system correctly.