INTRODUCTION TO DATA STRUCTURES

Evaluating postfix expressions & Translate infix to postfix in C

HW2_D84099084_賴澍雨

Shih Yu Lai

1.Evaluating postfix expressions

實作過程

當使用 C 語言編寫程式以利用堆疊(Stack)資料結構來評估postfix notation時，涉及堆疊這一抽象資料型態(ADT)以及評估postfix表示法的演算法。

使用的資料結構：堆疊(Stack)

堆疊是一種線性資料結構，遵循 後進先出 (LIFO)的原則。它有兩個主要操作：push(將一個項目加到頂部)和 pop(移除頂部項目)。透過堆疊實作中使用 Add(push)和 Delete(pop)函數來模擬這些操作。

程式邏輯與演算法

1. **初始化**：創建一個空堆疊。
2. **Tokenization**：使用空格作為 token，將表達式分解 為標記(operands 和operators)。
3. **處理tokens**：
   - 如果tokens是operands，將其轉換為整數並推入堆疊。
   - 如果tokens是operators，從堆疊彈出所需數量的operands，應用操作符，然後將結果推回堆疊。
4. **最終結果**：處理完所有 tokens後，表達式的最終結果是堆疊上剩下的單一元素。

測試正確性

簡單測試案例：使用您可以手動驗證結果的已知簡單表達式來運行程式。例如，"1 1 2 * +" 應該得到結果 3。
邊界測試案例：包括測試程式極限的表達式，例如包含大量tokens的表達式，或可能導致整數溢出的表達式。
錯誤處理：包含設計上會失敗的測試案例，例如具有太多operators的表達式(導致堆疊下溢)或太多 operands(可能會在最後留下多個項目在堆疊上)。

Shih Yu Lai

Stack

IsFull()

Add()

IsEmpty()

Delete()

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <limits.h> // Include for INT_MIN

#define MAX_STACK_SIZE 100

typedef struct {
    int items[MAX_STACK_SIZE];
    int top;
} Stack;

// Function to create a stack
Stack* CreateS() {
    Stack* S = (Stack*)malloc(sizeof(Stack)); // Allocate memory for the stack
    S->top = -1; // Initialize top to -1
    return S; // Return the stack
}

// Function to check if stack is full
bool IsFull(Stack* stack) {
    return stack->top == MAX_STACK_SIZE - 1; // Return true if stack is full
}

// Function to add item to stack
bool Add(Stack* stack, int item) {
    if (IsFull(stack)) {
        return false; // Return false if stack is full
    } else {
        stack->items[++stack->top] = item; // Add item to stack
        return true;
    }
}

// Function to check if stack is empty
bool IsEmpty(Stack* stack) {
    return stack->top == -1; // Return true if stack is empty
}

// Function to delete item from stack
int Delete(Stack* stack) {
    if (IsEmpty(stack)) {
        return INT_MIN; // or another value to indicate error
    } else {
        return stack->items[stack->top--]; // Delete item from stack
    }
}

// Function to evaluate postfix expressions
int EvaluatePostfix(char* expr) {
    Stack* stack = CreateS(); // Create stack
    char* token = strtok(expr, " "); // Initialize token with the first token

    while (token != NULL) {
        if (token[0] >= '0' && token[0] <= '9') { // If the token is an operand
            Add(stack, atoi(token));
        } else { // If the token is an operator
            int val2 = Delete(stack); // Get the second operand
            int val1 = Delete(stack); // Get the first operand
            switch (token[0]) {
                case '+': Add(stack, val1 + val2); break; // Add two operands and push the result to the stack
                case '-': Add(stack, val1 - val2); break; // Subtract two operands and push the result to the stack
                case '*': Add(stack, val1 * val2); break; // Multiply two operands and push the result to the stack
                case '/': Add(stack, val1 / val2); break; // Divide two operands and push the result to the stack
            }
        }
        token = strtok(NULL, " "); // Get the next token
    }

    int result = Delete(stack); // Get the result
    free(stack); // Don't forget to free the allocated memory
    return result; // Return the result
}

int main() {
    char expr[MAX_STACK_SIZE]; // Initialize the expression
    FILE *file = fopen("Q1.Testcase.txt", "r"); // Open the test file
    if (!file) {
        perror("Failed to open example.txt"); // Print error message
        return EXIT_FAILURE;
    }

    // Example
    char Example[] = "1 1 2 * +"; // Initialize the expression
    printf("Example:\n");
    printf("Input: %s, Output: %d\n", Example, EvaluatePostfix(Example)); // Print the result
    printf("-----------------------------------------------------------------\n");

    // Test
    file = fopen("Q1.Testcase.txt", "r"); // Open the test file
    if (!file) {
        perror("Failed to open test.txt"); // Print error message
        return EXIT_FAILURE;
    }

    printf("Test:\n");
    while (fgets(expr, MAX_STACK_SIZE, file) != NULL) { // Read each line
        expr[strcspn(expr, "\n")] = 0; // Remove newline character
        printf("Input: %s, Output: %d\n", expr, EvaluatePostfix(expr)); // Print the result
    }
    fclose(file); // Close the file

    return 0; // Return 0 to indicate successful completion
}
```

輸入範例
(簡報題目)

輸入測試
(Q1.Testcase)

測試與測試資料設計

正常案例：正確遵循postfix notation的表達式。
邊緣案例：僅有single number 或single operator的表達式，看程式如何處理最小的輸入。
無效輸入：提供具有無效tokens或語法的表達式，以確保程式可以正確處理錯誤。

實際測試

編譯並運行帶有一系列測試案例作為輸入的程式。然後將輸出與預期結果進行比較。
main 函數包括一個固定的測試案例陣列，代表了可能的postfix notation表達式的良好組合。為了確保堆疊操作本身被正確實現，為函數編寫測試。這些測試通過控制輸入調用函數，並斷言函數輸出或結果堆疊狀態是否如預期。其中包含:

1. 向堆疊中添加和刪除元素，在每次操作後檢查top元素。
2. 將堆疊填充到容量極限，以確保 IsFull 返回 true，並且 Add 不允許推入更多元素。
3. 從空堆疊中刪除元素，檢查是否有正確的錯誤處理。

輸出

```
問題  輸出  偵錯主控台  終端機  連接埠  CODE REFERENCE LOG  註解

PS D:\Data Structure> cd "d:\Data Structure\Hw2\" ; if ($?) { gcc Q1.c -o Q1 } ; if ($?) { .\Q1 }
Example:
Input: 1, Output: 3
-----------------------------------------------------------------
Test:
Input: 2, Output: 14
Input: 16, Output: 613
Input: 1, Output: 21
Input: 120, Output: 100
Input: 256, Output: 896
Input: 100000, Output: -12199
PS D:\Data Structure\Hw2>
```

2. Translate infix to postfix

實作過程(1/2)

C 語言程式利用堆疊資料結構將 infix notation（operator 位於operands之間的典型數學表示法）轉換為 postfix notation（operator 跟在operands後面）。

使用的資料結構：堆疊(Stack)

選擇堆疊作為這項任務的資料結構，因為它的後進先出（ LIFO）特性適合處理operator，並確保它們以正確的順序被應用。在處理表達式時，它暫時儲存operator 和括號。

輸出

```
問題  輸出  偵錯主控台  終端機  連接埠  CODE REFERENCE LOG  註解

PS D:\Data Structure> cd "d:\Data Structure\Hw2\" ; if ($?) { gcc Q2.c -o Q2 } ; if ($?) { .\Q2 }
----------------Example----------------
Infix: 1 + 1 * 2
Postfix: 1 1 2 * +
=======================================
----------------Test----------------
Infix: 2 + 3 * 4
Expected Postfix: 2 3 4 * +
Actual Postfix: 2 3 4 * +
Result: Correct
---------------------------------------
Infix: 16 * 38 + 5
Expected Postfix: 16 38 * 5 +
Actual Postfix: 16 38 * 5 +
Result: Correct
---------------------------------------
Infix: (1 + 2) * 7
Expected Postfix: 1 2 + 7 *
Actual Postfix: 1 2 + 7 *
Result: Correct
---------------------------------------
Infix: 120 * 5 / 6
Expected Postfix: 120 5 * 6 /
Actual Postfix: 120 5 * 6 /
Result: Correct
---------------------------------------
Infix: (256 / (10 - 6 + 12)) * (13 - 5) * 7
Expected Postfix: 256 10 6 - 12 + / 13 5 - * 7 *
Actual Postfix: 256 10 6 - 12 + / 13 5 - * 7 *
Result: Correct
---------------------------------------
Infix: 100000 / 20 - 135 + 26 * 76 -560 * 34
Expected Postfix: 100000 20 / 135 - 26 76 * + 560 34 * -
Actual Postfix: 100000 20 / 135 - 26 76 * + 560 34 * -
Result: Correct
---------------------------------------
PS D:\Data Structure\Hw2>
```

Data Structure > Hw2 > C Q2.c > infixToPostfix(char *, char *)

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>
#include <limits.h> // Include for INT_MIN

#define MAX_STACK_SIZE 100

// Stack structure and function declarations
typedef struct {
    int items[MAX_STACK_SIZE];
    int top;
} Stack;

Stack* CreateS();
bool Add(Stack* stack, int item);
int Delete(Stack* stack);
bool IsEmpty(Stack* stack);
bool IsFull(Stack* stack);
int precedence(char operator);
void infixToPostfix(char* infix, char* postfix);

// Implementations of stack functions
// Function to create a stack
Stack* CreateS() {
    Stack* S = (Stack*)malloc(sizeof(Stack)); // Allocate memory for stack
    S->top = -1;  // Initialize top to -1
    return S;
}

// Function to check if stack is full
bool IsFull(Stack* stack) {
    return stack->top == MAX_STACK_SIZE - 1; // Return true if stack is full
}

// Add an item to the stack
bool Add(Stack* stack, int item) {
    if (IsFull(stack)) {
        return false; // Return false if stack is full
    }
    stack->items[++stack->top] = item; // Add item to stack
    return true; // Return true if stack is not full
}

// Function to check if stack is empty
bool IsEmpty(Stack* stack) {
    return stack->top == -1; // Return true if stack is empty
}

// Function to delete an item from the stack
int Delete(Stack* stack) {
    if (IsEmpty(stack)) {
        return INT_MIN; // Use INT_MIN to indicate an error or empty stack
    }
    return stack->items[stack->top--];
}

// Function to return precedence of operators
int precedence(char operator) {
    switch (operator) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        default:
            return 0; // for non-operators
    }
}
```

```c
// Function to convert infix expression to postfix
void infixToPostfix(char* infix, char* postfix) {
    Stack* stack = CreateS();
    int i, j = 0;
    char token;
    for (i = 0; infix[i] != '\0'; i++) {
        token = infix[i];
        // Skip spaces in the infix expression
        if (token == ' ') {
            continue;
        }
        // Handle operands (digits)
        if (isdigit(token)) {
            // Add operand to postfix expression
            postfix[j++] = token;
            // Handle multi-digit numbers
            while (isdigit(infix[i + 1])) {
                postfix[j++] = infix[++i];
            }
            // Add space after operand
            postfix[j++] = ' ';
        } else if (token == '(') {
            // Push '(' onto stack
            Add(stack, token);
        } else if (token == ')') {
            // Pop everything up to the matching '('
            while (!IsEmpty(stack) && stack->items[stack->top] != '(') {
                postfix[j++] = Delete(stack);
                postfix[j++] = ' ';
            }
            // Pop the '('
            Delete(stack);
        } else {
            // Operator
            // Pop operators with higher or equal precedence
            while (!IsEmpty(stack) && precedence(token) <= precedence(stack->items[stack->top]) && stack->items[stack->top] != '(') {
                postfix[j++] = Delete(stack);
                postfix[j++] = ' ';
            }
            // Push the current operator onto stack
            Add(stack, token);
        }
    }
    // Pop any remaining operators from the stack
    while (!IsEmpty(stack)) {
        postfix[j++] = Delete(stack);
        postfix[j++] = ' ';
    }
    // Replace the last space with a null terminator
    if (j > 0 && postfix[j - 1] == ' ') {
        j--;
    }
    postfix[j] = '\0';
    // Free the stack
    free(stack);
}

// Main function to read infix expressions from a file and convert them to postfix
int main() {
    printf("----------------Example----------------\n");
    char exampleInfix[] = "1 + 1 * 2"; // Example infix expression
    char examplePostfix[strlen(exampleInfix) + 1]; // Postfix expression buffer
    infixToPostfix(exampleInfix, examplePostfix); // Convert infix expression to postfix
    printf("Infix: %s\nPostfix: %s\n", exampleInfix, examplePostfix); // Print the postfix expression

    printf("=======================================\n");

    // Test
    printf("----------------Test----------------\n");
    FILE *q2File = fopen("Q2.Testcase.txt", "r"); // Open Q2 test file for reading
    FILE *q1File = fopen("Q1.Testcase.txt", "r"); // Open Q1 test file for reading
    char infix[MAX_STACK_SIZE]; // Buffer for reading an infix expression
    char postfix[MAX_STACK_SIZE]; // Buffer for the corresponding postfix expression
    char expectedPostfix[MAX_STACK_SIZE]; // Buffer for the expected postfix expression from Q1

    if (q2File != NULL && q1File != NULL) {
        while (fgets(infix, sizeof(infix), q2File)) {
            if(fgets(expectedPostfix, sizeof(expectedPostfix), q1File) == NULL) {
                printf("Error reading from Q1 Testcase file.\n");
                break;
            }
            infix[strcspn(infix, "\n\r")] = 0; // Remove newline character
            expectedPostfix[strcspn(expectedPostfix, "\n\r")] = 0; // Remove newline character

            infixToPostfix(infix, postfix); // Convert infix to postfix
            printf("Infix: %s\nExpected Postfix: %s\nActual Postfix: %s\n", infix, expectedPostfix, postfix); // Print the results

            // Compare actual postfix with expected postfix
            if (strcmp(postfix, expectedPostfix) == 0) {
                printf("Result: Correct\n");
            } else {
                printf("Result: Incorrect\n");
            }
            printf("---------------------------------------\n");
        }
        fclose(q2File); // Close Q2 test file
        fclose(q1File); // Close Q1 test file
    } else {
        if (q2File == NULL) perror("Error opening Q2 Testcase file");
        if (q1File == NULL) perror("Error opening Q1 Testcase file");
    }

    return 0; // Return 0 to indicate successful execution
}
```

輸入

- Q2.Testcase是infix,Q1.Testcase是postfix
- 用Q2.Testcase為輸入，Q1.Testcase作為正解檢查Q2.Testcase的output

輸入範例
(簡報題目)

輸入測試
(Q2.Testcase)

測試

2. Translate infix to postfix

實作過程(2/2)

程式邏輯與演算法

1. **創建堆疊:** 在轉換過程中用來儲存 operator 。
2. **對infix notation進行tokenize** :逐字元處理 infix notation。
3. **處理每個字元:**
    - 如果是數字，它是 operands的一部分，因此直接加入到 postfix字符串。
    - 如果是開括號 (，將其壓入堆疊。
    - 如果是閉括號)，從堆疊中彈出直到彈出一個開括號，並將彈出的 operator 加到postfix字符串。
    - 如果是operator，彈出堆疊中優先級更高或相等的 operator，並將它們加入 postfix字符串，然後將當前 operator 壓入堆疊。
4. **清空堆疊:** 在表達式的結尾，將堆疊中剩餘的 operator 彈出至postfix字符串。
5. **處理空白:** 空白不視為 postfix notation的一部分，將被跳過。

轉換所用的演算法是 Shunting Yard algorithm，該演算法專為解析 infix記號法指定的數學表達式而設計。它可用於 產生Reverse Polish notation (postfix)和abstract syntax trees。

測試正確性

為了驗證程式的正確性，使用了兩個檔案： Q1.Testcase.txt 包含postfix notation，和 Q2.Testcase.txt 包含對應的infix notation。

1. **運行示例案例** :首先運行一個例子來演示功能。
2. **從 Q2.Testcase.txt 讀取測試案例** :程式讀取每個 infix notation。
3. **轉換為postfix**:然後使用實作的函數將它們轉換為 postfix notation。
4. **從 Q1.Testcase.txt 讀取預期結果** :程式從第一個文件讀取預期的 postfix結果。
5. **與實際輸出比較** :將轉換結果與預期結果進行比較。
6. **Print結果**:打印每個測試案例的結果，指出轉換是否與預期的 postfix notation匹配。

測試與測試資料設計

簡單案例:單個數字和無括號的簡單表達式。
複雜案例:具有多位數字、多個操作和括號的表達式。
邊緣案例:測試優先規則的表達式，例如 "3 + 4 * 2" 應該結果為 "3 4 2 * +"。
通過將程式輸出與預期的postfix notation進行比較，我們確保堆疊操作和infix轉postfix的轉換邏輯正常運作。

實際測試

運行程式時，確保兩個測試檔案（Q1.Testcase.txt 和 Q2.Testcase.txt）位於可執行檔的正確位置，並且它們的內容根據infix和postfix表示規則正確格式化。主函數中的比較將確認轉換演算法是否按預期工作。