

数据结构与算法 I 实验报告

实验 1：栈和队列的应用

龚舒凯 2022202790 应用经济-数据科学实验班

<https://github.com/GONGSHUKAI>

2023 年 9 月 29 日

算数表达式求值演示

1 需求分析

问题描述：表达式计算实现程序设计语言的基本问题之一，也是栈的应用的一个典型例子。设计一个程序，演示用算符优先法对算术表达式求值的过程。

基本要求：以字符序列的形式从终端输入语法正确、不含变量的整数表达式。利用课件给出的算符优先关系，实现对算术四则混合运算表达式的求值，并演示在求值过程中操作符栈、操作数栈、输入字符和主要操作的变化过程。

输入形式：一个以“=”结尾的中缀算术表达式。运算符包括 +, -, *, /, (,), =, ^, 参加运算的数为正整数。例如， $3*(7-2)$ 或者 $1024/(20+8)$ 或者 $(20+2)*(6*(2+8))$ 或者 $((2+4)*(5+7)*9+1)/2$ 等等。

输出形式：表达式的计算结果。运算过程中操作符栈、操作数栈、输入字符和主要操作的内容。

2 概要设计

中缀表达式的求值需要用到两个栈：操作符栈OPTR，操作数栈OPND。操作数栈存放的数据为double类型，而操作符栈存放的数据为char类型。

2.1 抽象数据类型

在本程序中，用到的抽象数据类型定义如下所示：

```
ADT char_stack{
    数据对象：  $D = \{c | c \in \text{char}\}$ 
    数据关系：  $R = \{\langle e_1, e_2 \rangle | e_1 \text{ 是一个 char [] 类型的数组, } e_2 \text{ 是一个指向栈顶的指针}\}$ 
    基本操作：
        InitCharStack() 操作结果：创建一个字符串类型的栈
        CharStackPush(s, value) 操作结果：入栈字符 s
        CharStackPop(s) 操作结果：将栈顶元素弹出
        CharStackPrint(s) 操作结果：打印当前栈中所有元素
}
ADT stack{
    数据对象：  $D = \{c | c \in \text{double}\}$ 
    数据关系：  $R = \{\langle e_1, e_2 \rangle | e_1 \text{ 是一个 double [] 类型的数组, } e_2 \text{ 是一个指向栈顶的指针}\}$ 
    基本操作：
        InitStack() 操作结果：创建一个字符串类型的栈
        StackPush(s, value) 操作结果：入栈字符 s
        StackPop(s) 操作结果：将栈顶元素弹出
        StackPrint(s) 操作结果：打印当前栈中所有元素
}
```

2.2 主程序流程

1. 首先创建操作符栈OPTR和操作数栈OPND
2. 使用gets()函数获取终端输入的字符串
3. 调用自创建函数InfixCalculation(str[], OPND, OPTR)计算中缀表达式

3 详细设计

两种栈的基本操作在附录中有详细代码实现。这里主要论述中缀表达式计算程序中三个核心函数的详细设计。

3.1 中缀表达式计算函数

中缀表达式计算函数的定义如下：

```
void InfixCalculation(char InfixExpression[], stack *OPND, char_stack *OPTR);
```

其中 `char InfixExpression[]` 是终端输入的中缀表达式；`stack *OPND` 是操作数栈；`stack *OPTR` 是操作符栈。算法思想如下：

1. 建立并初始化 `OPTR` 栈和 `OPND` 栈，然后在 `OPTR` 栈中压入一个 `=`
2. 扫描中缀表达式，取一字符送入 `c`
3. 当 `c == '='` 且 `OPTR` 栈的栈顶 `= '='` 时才停止循环，在 `OPTR` 栈的栈顶得到运算结果，否则执行以下操作：
 - (a) 若 `c` 是操作数，进 `OPND` 栈，从中缀表达式取读下一字符送入 `c`
 - (b) 若 `c` 是操作符，比较 `icp(ch)` 的优先级（栈外优先数）和 `isp(OPTR)` 的优先级（栈内优先数）：
 - i. 若 `icp(ch) > isp(OPTR)`，则 `c` 进 `OPTR` 栈，从中缀表达式取下一字符送入 `c`
 - ii. 若 `icp(ch) < isp(OPTR)`，则从 `OPND` 栈退出两个元素 `a2` 和 `a1`，从 `OPTR` 栈退出 `X`，形成运算指令 `(a1)X(a2)`，结果进 `OPND` 栈；
 - iii. 若 `icp(c) = isp(OPTR)` 且 `c == ')''`，则从 `OPTR` 栈退出 `c == '('`，对消括号，然后从中缀表达式取下一字符送入 `c`

其中，算数操作符的优先级表如下表所示

操作符 <code>c</code>	栈内优先数 <code>isp</code>	栈外优先数 <code>icp</code>
<code>#</code>	0	0
<code>(</code>	1	6
<code>*,/,%</code>	5	4
<code>+, -</code>	3	2
<code>)</code>	6	1

表 1: 算数操作符优先级表

代码如下所示：

```
1 void InfixCalculation(char InfixExpression[], stack *OPND, char_stack *OPTR){
2     CharStackPush(OPTR, '=');//先往 OPTR 栈中压入一个 #
3     while(1){//扫描中缀表达式的各个字符
4         if (IsOperand(InfixExpression[i]) == TRUE){//如果扫描到操作数
```

```

5         double num = Digitizer(InfixExpression); //Digitizer 是将字符转换为数字的函数
6         StackPush(OPND, num); //将该操作数入栈
7         std::cout << " 当前操作符栈为: ";
8         CharStackPrint(OPTR);
9         std::cout << " 当前操作数栈为: ";
10        StackPrint(OPND);
11        std::cout << std::endl;
12    }
13    else if (IsOperator(InfixExpression[i]) == TRUE){ //如果扫描到操作符
14        JudgePriority(InfixExpression[i], OPND, OPTR);
15        //JudgePriority 是用于判断当前操作符和栈顶操作符优先级的函数
16    }
17    if (OPTR->data[OPTR->top] == '=' && InfixExpression[i] == '=') break; //终止循环的条件
18    i++; //移动到下一个字符的位置
19 }
20 std::cout << OPND->data[OPND->top]; //栈顶元素即为中缀表达式的计算值
21 }

```

3.2 字符-数字转换函数

字符-数字转换函数的定义如下

```
int Digitizer(char str[]);
```

其中`str[]`表示输入的中缀表达式。函数的功能是将扫描的操作数（`char`形式）转换为数字（`double`形式）。由于限定了参加运算的数为正整数，因此考虑下述两种情况：

1. 运算数为正的一位数，如 0,1,...。在中缀表达式中，这样的运算数后必然是运算符
2. 运算数为正的多位数，如 1024。

我们可以这样区分一位数与多位数：首先设置标记变量`sign = 0`。当扫描到中缀表达式的某个单字符`str[i]`时，我们将其转化为数字`x`，并置标记`sign = 1`。在扫描下一个单字符时，如果仍然扫到数字，则表明运算数为多位数，通过

$$x = x * 10 + (\text{double})\text{str}[i+1]$$

实现位数的拼接；如果扫到操作符，则表明运算数为单位数，`Digitizer(str)`函数直接返回`x`值。

代码如下所示：

```

1 int Digitizer(char str[]){
2     double x = 0; //要返回的数字
3     int sign = 0; //标识变量，用于将字符串转换为十位数、百位数
4     while (IsOperand(str[i]) == TRUE){
5         if (sign != 1){

```

```

6         x = char_to_double(str[i]);
7         sign = 1;
8         //把这个字符转换为数字后, 置 sign = 1
9         //如果这是一位数, 则 str[i+1] 必然不是数字
10        //如果这是多位数, 则 str[i+1] 仍然是数, 由下面的语句将各个位数拼接成多位数
11    }
12    else{
13        x = x * 10 + char_to_double(str[i]);
14    }
15    i++;
16 }
17 i--;
18 return x;
19 }

```

3.3 优先级判断函数

优先级判断函数的定义如下

```
void JudgePriority (char c, stack *OPND, char_stack *OPTR);
```

该函数用于判断当前算数操作符c和栈顶操作符OPTR->data[OPTR->top]优先级的函数。该函数依托表1实现, 代码如下所示:

```

1 void JudgePriority (char c, stack *OPND, char_stack *OPTR){
2     //根据算数运算符优先级表制作以下对应法则
3     //OPND 是操作数栈
4     //OPTR 是操作符栈
5     int isp = 0; //栈内优先数
6     int icp = 0; //栈外优先数
7     if (c == '=') icp = 0;
8     else if (c == '(') icp = 6;
9     else if (c == '*' || c == '/' || c == '%') icp = 4;
10    else if (c == '+' || c == '-') icp = 2;
11    else icp = 1;
12
13    if (OPTR->data[OPTR->top] == '=') isp = 0;
14    else if (OPTR->data[OPTR->top] == '(') isp = 1;
15    else if (OPTR->data[OPTR->top] == '*' || OPTR->data[OPTR->top] == '/' ||
16             OPTR->data[OPTR->top] == '%') isp = 5;
17    else if (OPTR->data[OPTR->top] == '+' || OPTR->data[OPTR->top] == '-') isp = 3;
18    else isp = 6;
19 }

```

```
20     if (icp > isp){
21         //如果 icp(c) > isp(OPTR), 则 c 进 OPTR 栈
22         //从中缀表达式中取下一字符送入 c
23         CharStackPush(OPTR, c);
24         std::cout << " 当前操作符栈为: ";
25         CharStackPrint(OPTR);
26         std::cout << " 当前操作数栈为: ";
27         StackPrint(OPND);
28         std::cout << std::endl;
29     }
30     else if (icp < isp){
31         //如果 icp(c) < isp(OPTR)
32         //OPND 栈退出 a2 和 a1, 从 OPTR 栈退出 , 形成运算指令 (a1) (a2)
33         //结果压入 OPND 栈
34         double a2 = StackTop(OPND);
35         StackPop(OPND);
36         double a1 = StackTop(OPND);
37         StackPop(OPND);
38         char this_OPTR = CharStackTop(OPTR);
39         double ans = Calculate(a1, a2 ,this_OPTR);
40         std::cout << " 当前执行操作: " << a1 << this_OPTR << a2 << '=' << ans << std::endl;
41         StackPush(OPND,ans); //运算结果压入 OPND 栈
42         CharStackPop(OPTR); //运算符 退出 OPTR 栈
43         std::cout << " 当前操作符栈为: ";
44         CharStackPrint(OPTR);
45         std::cout << " 当前操作数栈为: ";
46         StackPrint(OPND);
47         std::cout << std::endl;
48
49         JudgePriority(c, OPND, OPTR);
50     }
51     else{
52         //果 icp(c) = isp(OPTR), 有两种情况:
53         //如果 c = '=' 则整个运算结束
54         //如果 c = ')' 则需要从 OPTR 中弹出 '(' 从而对消括号
55         if (OPTR->data[OPTR->top] == '=') return;
56         else{
57             CharStackPop(OPTR);
58             std::cout << " 当前操作符栈为: ";
59             CharStackPrint(OPTR);
60             std::cout << " 当前操作数栈为: ";
61             StackPrint(OPND);
```

```
62         std::cout << std::endl;
63     }
64 }
65 }
```

注意到对于`else if (icp < isp)`所在的分支，当栈外优先级 $<$ 栈内优先级时，需要不断将OPTR中的运算符弹出，直到栈外优先级 \geq 栈内优先级，才能读入下一个操作符。

因此，我们采用递归调用的处理方法，递归调用`JudgePriority(c, OPND, OPTR)`，直到`icp = isp`或`icp > isp`才结束递归，读入新的操作符`c`。

4 用户使用说明

程序运行后，在终端输入中缀算数表达式。输入格式有以下要求：

- 中缀表达式必须以`=`结尾。
- 中缀表达式间不得有“非运算符”、“非运算数”，如空格、`#`、`&` 等。`114514 * (1919 - 810) =` 是不合法的输入。
- 中缀表达式必须是可计算的。`((2+4)*(5+7)*9+1)/2=`是不合法的输入，因为左侧多了一个`(`。
- 中缀表达式中的运算数必须为正整数。不能输入变量（如 `a, b, e`），负数，小数等。

5 测试结果

有以下测试样例供参考：

输入：3*(7-2)=	输出：15
输入：1024/(20+8)=	输出：36.5714
输入：(20+2)*(6*(2+8))=	输出：1320
输入：((2+4)*(5+7)*9+1)/2=	输出：324.5

以第四组测试结果为例：

1	当前操作符栈为：= (
2	当前操作数栈为：
3	
4	当前操作符栈为：= ((
5	当前操作数栈为：
6	
7	当前操作符栈为：= ((
8	当前操作数栈为：2
9	
10	当前操作符栈为：= ((+
11	当前操作数栈为：2
12	
13	当前操作符栈为：= ((+
14	当前操作数栈为：2 4
15	
16	当前执行操作：2+4=6
17	当前操作符栈为：= ((
18	当前操作数栈为：6
19	
20	当前操作符栈为：= (
21	当前操作数栈为：6
22	
23	当前操作符栈为：= (*
24	当前操作数栈为：6
25	
26	当前操作符栈为：= (* (
27	当前操作数栈为：6
28	
29	当前操作符栈为：= (* (
30	当前操作数栈为：6 5
31	
32	当前操作符栈为：= (* (+

33 当前操作数栈为：6 5

34

35 当前操作符栈为：= (* (+

36 当前操作数栈为：6 5 7

37

38 当前执行操作：5+7=12

39 当前操作符栈为：= (* (

40 当前操作数栈为：6 12

41

42 当前操作符栈为：= (*

43 当前操作数栈为：6 12

44

45 当前执行操作：6*12=72

46 当前操作符栈为：= (

47 当前操作数栈为：72

48

49 当前操作符栈为：= (*

50 当前操作数栈为：72

51

52 当前操作符栈为：= (*

53 当前操作数栈为：72 9

54

55 当前执行操作：72*9=648

56 当前操作符栈为：= (

57 当前操作数栈为：648

58

59 当前操作符栈为：= (+

60 当前操作数栈为：648

61

62 当前操作符栈为：= (+

63 当前操作数栈为：648 1

64

65 当前执行操作：648+1=649

66 当前操作符栈为：= (

67 当前操作数栈为：649

68

69 当前操作符栈为：=

70 当前操作数栈为：649

71

72 当前操作符栈为：= /

73 当前操作数栈为：649

74

75 当前操作符栈为： = /
76 当前操作数栈为： 649 2
77
78 当前执行操作： 649/2=324.5
79 当前操作符栈为： =
80 当前操作数栈为： 324.5

6 调试分析

6.1 算法的时空分析

1. 时间复杂度:

显然,我们必须遍历中缀表达式的每个元素才能计算出中缀表达式的值,故共执行 n 次操作,其中 n 是中缀表达式的长度。

对于每个中缀表达式的元素,即使在最坏的情况下,我们执行的一系列操作的时间复杂度都是常数级的(如:入栈运算符、入栈运算数、弹出运算符执行有限次、字符-数字转换执行有限次等)。因此,整个算法的时间复杂度是 $O(n)$ 。

2. 空间复杂度:

我们一共使用了两个栈OPTR,OPND,栈的最大大小取决于中缀表达式的结构。在最坏情况下,栈的大小可能与中缀表达式的长度成正比。因此,整个算法的空间复杂度是 $O(n)$ 。

7 附录

详细的代码实现如下所示。也可以通过https://github.com/GONGSHUKAI/Data_Structure/tree/main/Lab_Code/Lab_1/Sept.22_Lab下载代码原文件。

```
#include <iostream>
#include <cmath>
//#define OVERFLOW -2
//#define UNDERFLOW -1
#define TRUE 1
#define FALSE 0
#define MAX 100//栈的大小
#define MAXLENGTH 100//中缀表达式的大小

typedef struct char_stack{
    char data[MAX];
    int top;
}char_stack;

typedef struct stack{
    double data[MAX];
    int top;
}stack;

int i = 0;

char_stack* InitCharStack();
void CharStackPush(char_stack *s, char value);//入栈
void CharStackPop(char_stack *s);//出栈
void CharStackPrint(char_stack *s);//打印当前栈内所有元素
int IsOperand (char s);//判断是否为操作数
int IsOperator (char s);//判断是否为操作符
void JudgePriority (char c,stack *OPND, char_stack *OPTR);//判断算数操作符的优先级
int Digitizer(char str[]);

stack* InitStack();
void DestroyStack(stack *s);
int StackEmpty(stack *s);
double StackTop(stack *s);
void StackPush(stack *s, double value);
void StackPop(stack *s);
int StackSize(stack *s);
void StackPrint(char_stack *s);//打印当前栈内所有元素
```

```
double char_to_double(char s);
double Calculate (double x, double y, char s);

void InfixCalculation(char InfixExpression[], stack *OPND, char_stack *OPTR);

int main(){
    stack *OPND = InitStack();
    char_stack *OPTR = InitCharStack();

    char InfixExpression[MAXLENGTH];
    gets(InfixExpression);
    InfixCalculation(InfixExpression, OPND, OPTR);
}

stack* InitStack(){
    stack *s = new stack;
    s->top = -1; //栈顶指针（即数组下标）赋初值-1
    return s;
}

//栈底下标是 0，栈顶下标最多到 MAX-1
void DestroyStack(stack *s){
    delete s;
}

int StackEmpty(stack *s){ //如果栈空返回 1，否则返回 0
    if (s->top == -1) return TRUE;
    else return FALSE;
}

double StackTop(stack *s){ //返回栈顶元素
    if (s->top > -1 && s->top < MAX) return s->data[s->top];
    else return FALSE;
}

void StackPush(stack *s, double value){ //压入栈
    if (s->top == MAX - 1) return;
    else s->data[++s->top] = value;
}

void StackPop(stack *s){ //弹出栈顶元素
    if (s->top == -1) return;
```

```
        else{
            s->top--;
        }
    }

    int StackSize(stack *s){//返回栈的大小
        return s->top++;
    }

    void StackPrint(stack *s){//打印当前栈内所有元素
        for (int i = 0 ; i <= s->top ; i++){
            std::cout << s->data[i] << " ";
        }
        std::cout << std::endl;
    }

    double char_to_double(char s){
        if (s == '0') return 0;
        else if (s == '1') return 1;
        else if (s == '2') return 2;
        else if (s == '3') return 3;
        else if (s == '4') return 4;
        else if (s == '5') return 5;
        else if (s == '6') return 6;
        else if (s == '7') return 7;
        else if (s == '8') return 8;
        else return 9;
    }

    double Calculate (double x, double y, char s){
        if (s == '+') return x+y;
        else if (s == '-') return x-y;
        else if (s == '*') return x*y;
        else if (s == '/') return x/y;
        else return std::pow(x,y);
    }

    char_stack* InitCharStack(){
        char_stack *s = new char_stack;
        s->top = -1;//栈顶指针（即数组下标）赋初值-1
        return s;
    }
```

```
char CharStackTop (char_stack *s){
    if (s->top > -1 && s->top < MAX) return s->data[s->top];
    else return FALSE;
}

void CharStackPush(char_stack *s, char value){//压入栈
    if (s->top == MAX - 1) return;
    else s->data[++s->top] = value;
}

void CharStackPop(char_stack *s){
    if (s->top == -1) return;
    else{
        s->top--;
    }
}

void CharStackPrint(char_stack *s){//打印当前栈内所有元素
    for (int i = 0 ; i <= s->top ; i++){
        std::cout << s->data[i] << " ";
    }
    std::cout << std::endl;
}

int IsOperand (char s){
    if (s != '(' && s != ')' && s != '+' && s != '-' &&
        s != '*' && s != '/' && s != '%' && s != '^' && s != '='){
        return TRUE;
    }
    else return FALSE;
}

int IsOperator (char s){
    if (s == '(' || s == ')' || s == '+' || s == '-' ||
        s == '*' || s == '/' || s == '%' || s == '^' || s == '='){
        return TRUE;
    }
    else return FALSE;
}

int Digitizer(char str[]){
```



```

double x = 0; //要返回的数字
int sign = 0; //标识变量, 用于将字符串转换为十位数、百位数
int negative_sign = 0; //标识变量, 用于将字符串转换为负数
while (IsOperand(str[i]) == TRUE){
    if (sign != 1){
        x = char_to_double(str[i]);
        sign = 1;
        //把这个字符转换为数字后, 置 sign = 1
        //如果这是一位数, 则 str[i+1] 必然不是数字
        //如果这是多位数, 则 str[i+1] 仍然是数, 由下面的语句将各个位数拼接成多位数
    }
    else{
        x = x * 10 + char_to_double(str[i]);
    }
    i++;
}
i--;
return x;
}

void JudgePriority (char c, stack *OPND, char_stack *OPTR){
    //s1 是操作数栈
    //s2 是操作符栈
    int isp = 0; //栈内优先数
    int icp = 0; //栈外优先数
    if (c == '=') icp = 0;
    else if (c == '(') icp = 8;
    else if (c == '^') icp = 6;
    else if (c == '*' || c == '/' || c == '%') icp = 4;
    else if (c == '+' || c == '-') icp = 2;
    else icp = 1;

    if (OPTR->data[OPTR->top] == '=') isp = 0;
    else if (OPTR->data[OPTR->top] == '(') isp = 1;
    else if (OPTR->data[OPTR->top] == '^') isp = 7;
    else if (OPTR->data[OPTR->top] == '*' || OPTR->data[OPTR->top] == '/' || OPTR->data[OPTR->top] == '%') isp = 3;
    else if (OPTR->data[OPTR->top] == '+' || OPTR->data[OPTR->top] == '-') isp = 3;
    else isp = 8;

    if (icp > isp){
        //如果 icp(c) > isp(OPTR), 则 c 进 OPTR 栈
        //从中缀表达式中取下一字符送入 c
    }
}

```

```

        CharStackPush(OPTR, c);
        std::cout << " 当前操作符栈为: ";
        CharStackPrint(OPTR);
        std::cout << " 当前操作数栈为: ";
        StackPrint(OPND);
        std::cout << std::endl;
    }
    else if (icp < isp){
        //如果 icp(c) < isp(OPTR)
        //OPND 栈退出 a2 和 a1, 从 OPTR 栈退出 , 形成运算指令 (a1) (a2)
        //结果压入 OPND 栈
        double a2 = StackTop(OPND);
        StackPop(OPND);
        double a1 = StackTop(OPND);
        StackPop(OPND);
        char this_OPTR = CharStackTop(OPTR);
        double ans = Calculate(a1, a2 ,this_OPTR);
        std::cout << " 当前执行操作: " << a1 << this_OPTR << a2 << '=' << ans << std::endl;
        StackPush(OPND,ans);//运算结果压入 OPND 栈
        CharStackPop(OPTR);//运算符 退出 OPTR 栈
        std::cout << " 当前操作符栈为: ";
        CharStackPrint(OPTR);
        std::cout << " 当前操作数栈为: ";
        StackPrint(OPND);
        std::cout << std::endl;

        JudgePriority(c, OPND, OPTR);
    }
    else{
        //果 icp(c) = isp(OPTR), 有两种情况:
        //如果 c = '=' 则整个运算结束
        //如果 c = ')' 则需要从 OPTR 中弹出 '('从而对消括号
        if (OPTR->data[OPTR->top] == '=') return;
        else{
            CharStackPop(OPTR);
            std::cout << " 当前操作符栈为: ";
            CharStackPrint(OPTR);
            std::cout << " 当前操作数栈为: ";
            StackPrint(OPND);
            std::cout << std::endl;
        }
    }
}

```

```
}
```

```
void InfixCalculation(char InfixExpression[], stack *OPND, char_stack *OPTR){
    CharStackPush(OPTR, '=');
    while(1){
        if (IsOperand(InfixExpression[i]) == TRUE){
            double num = Digitizer(InfixExpression);
            //double num = char_to_double(InfixExpression[i]);
            StackPush(OPND, num);
            std::cout << " 当前操作符栈为: ";
            CharStackPrint(OPTR);
            std::cout << " 当前操作数栈为: ";
            StackPrint(OPND);
            std::cout << std::endl;
        }
        else if (IsOperator(InfixExpression[i]) == TRUE){
            JudgePriority(InfixExpression[i], OPND, OPTR);
        }
        if (OPTR->data[OPTR->top] == '=' && InfixExpression[i] == '=') break;
        i++;
    }
    std::cout << OPND->data[OPND->top];
}
```
