北京師範大學
BEIJING NORMAL UNIVERSITY

《数据结构》上机实验报告

第 2 次上机

学号：202011140104

姓名：李馨

学院：物理学系

专业：物理学

教师：郑新

日期：2022.9.16

一、实验要求

1. 上机之前应做好充分准备，认真思考所需的上机题目，提高上机效率。

2. 独立上机输入和调试自己所编的程序，切忌抄袭、拷贝他人程序。

3. 上机结束后，整理出实验报告。书写报告时，重点放在实验的方法、思路以及总结反思上，以达到巩固课堂学习、提高动手能力的目的。

二、实验过程

a) 问题描述：合理运用栈，按照教材中的运算优先级，编程实现任意中缀算术表达式（可以只包含’+’、’-‘、’*’、’/’等双目运算符、小括号和结束符）的求值运算。

b) 问题分析与关键代码

i. **中缀表达式转化为后缀表达式**（Infix_to_Postfix.cpp）

直接使用中缀表达式计算，需要用到两个栈：操作符栈 OPTR (operator)，操作数栈 OPND(operand)。使用后缀表达式计算时需要一个操作数栈。这里我们考虑先将前缀表达式转化为后缀表达式，然后利用后缀表达式计算，这一步也需要用到一个栈。

1. 对中缀表达式进行扫描，读到一个操作数时，将之输出到结果字符串中；读到操作符时不立即输出，根据情况存到栈中，等待输出处理。
   a) 当读到左括号时，无论何时都直接存到栈中，无需其他操作。
      i. 左括号当且仅当处理右括号时会弹出，不会输出。
   b) 当读到右括号时，不将之存到栈中（意味着不会被输出），将栈顶元素弹出并输出，直到弹完左括号，左括号弹出但不输出。
   c) 当读到其他操作符时，暂时不将之存到栈中，将栈顶优先级更高的元素弹出并输出，直到遇到优先级比该读入字符低的元素；如果遇到和读入字符相同的操作符，也进行退栈和输出操作。退栈完成后将此操作符压入栈中。（这意味着栈中双目运算符的优先级从底到顶是递增的。）
   d) 如果读到输入的中缀表达式的末尾，将栈顶元素弹出并输出，直到变成空栈。
      i. 为了判断表达式末尾我们设立结束符“#”。
      ii. 读入'#'时进行的操作与读到右括号进行的操作（括号配对前不断退栈输出）有相似之处，为统一处理，初始时在栈底压入“#”。
2. 为实现上述算法的运行，我们设立如下所示的算术操作符的栈外优先级（icp）和栈内优先级（isp）：

	#	+ -	* /	(	)
icp	0	2	4	6	1
isp	0	3	5	1	6

a) 为了处理 1(a)和 1(b)的情况，将左括号的栈外优先级 icp 设得足够高，右括号栈外优先级 icp 足够低；为了处理栈外双目操作符与栈内相同时退栈输出的情况，双目运算符 isp 都大于 icp。

b) 操作符优先数相等的情况只出现在括号配对或栈底的“#”号与输入流最后的“#”号配对时。

3. 返回优先级的函数：

```
// 栈内优先级in stack priority
int isp(char ch) {
    switch(ch) {
        case '#': return 0; break;
        case '+': return 3; break;
        case '-': return 3; break;
        case '*': return 5; break;
        case '/': return 5; break;
        case '%': return 5; break;
        case '(': return 1; break;
        case ')': return 6; break;
        default: break;
    }
    exit(1);
}

// 栈外优先级in coming priority
int icp(char ch) {
    switch(ch) {
        case '#': return 0; break;
        case '+': return 2; break;
        case '-': return 2; break;
        case '*': return 4; break;
        case '/': return 4; break;
        case '%': return 4; break;
        case '(': return 6; break;
        case ')': return 1; break;
        default: break;
    }
    exit(1);
}
```

4. 算法实现：

a) 操作符栈 OPTR 初始化，将结束符“#”进栈，op 取栈顶元素。然后读入中缀表达式字符串的首字符 ch。

```
void InToPost(char A[], char *result) {
    SeqStack<char> OPTR; OPTR.Push('#'); // 操作符栈初始化，结束符'#'进栈
    int i = 0, j = 0; // j是A的索引，用于读取；i是result的索引，用于往里储存
    char ch = A[0]; // 开始扫描
    char op = OPTR.getTop(); // op取栈顶元素
```

b) 重复执行以下步骤，直到 ch = ‘#’，同时栈顶操作符也为‘#’，停止循环。

i. 若 ch 是操作数，直接输出，读入下一个字符 ch。

```
while (ch != '#' || op != '#') {
    if ((ch >= '0' && ch <= '9') || (ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z')) {
        // 若ch是操作数，直接输出，读入下一字符
        result[i] = ch; i += 1;
        j += 1; ch = A[j];
    }
```

ii. 若 ch 是操作符，判断 icp(ch)和 isp(op)：

1. 若 icp（ch）> isp（op），令 ch 进栈，读入下一个字符 ch。

```
if (icp(ch) > isp(op)) {
    // 栈外优先级高，进栈，读入下一字符
    OPND.Push(ch); op = OPND.getTop();
    j += 1; ch = A[j];
}
```

2. 若 icp（ch）< isp（op），退栈并输出。

```
else if (icp(ch) < isp(op)) {
    // 栈内优先级高，退栈并输出
    result[i++] = OPTR.Pop();
    op = OPTR.getTop();
}
```

3. 若 icp（ch）== isp（op），退栈但不输出，若退出的是“(”号读入下一个字符 ch。

```
else if (icp(ch) == isp(op)) {
    // 栈内栈外优先级相同，退栈但不输出（只可能左括号和右括号）
    // 如果退出的是'('，读入下一字符
    temp = OPTR.Pop(); op = OPTR.getTop();
    if (temp == '(') {
        j += 1;
        ch = A[j];
    }
}
```

iii. 为了用输出的字符串计算后缀表达式，在输出的字符串末尾加上‘#’。

```
result[i] = '#';
```

ii. 利用栈对后缀表达式求值（calculateRPN.cpp）

1. 初始化操作数栈 OPND，开始用 ch 扫描输入的字符串 A。

```
double calculateRPN(char A[]) {
    // 对以字符"#"结束的后缀表达式字符串进行计算。如果不是可计算的后缀表达式，显示出错信息并退出
    /* 顺序扫描表达式的每一项，然后根据它的类型做如下相应操作，
    如果该项是操作数，则将其压入栈中；
    如果该项是操作符<op>，则连续从栈中退出两个操作数Y和X，形成运算指令X<op>Y，并将计算结果重新压入栈中。
    当表达式的所有项都扫描并处理完后，栈顶存放的就是最后的计算结果。*/
    SeqStack<double> OPND;    // 存放操作数和计算结果的栈
    double result; int i = 0; double Y, X;  //结果数，字符串扫描指针，两个存放退栈的操作数的变量
    char ch = A[i];
```

2. 顺序扫描后缀表达式（以“#”结尾）的每一项，然后根据它的类型做如下相应操作，

a) 如果该项是操作数，则将其压入栈中；

```
while (ch != '#') {
    if (ch >= '0' && ch <= '9') {
        // 读到操作数，进栈
        OPND.Push((double)(ch - '0'));
    }
```

b) 如果该项是操作符<op>，则连续从栈中退出两个操作数 Y 和 X，形成运算指令 X<op>Y，并将计算结果重新压入栈中。

```
else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
    // 读到操作符，退栈两个操作数进行运算
    if (OPND.top < 1) {
        // 栈中不够两个操作数
        printf("可能输入了错误的后缀表达式，无法运算！\n"); exit(1);
    }
    Y = OPND.Pop(); X = OPND.Pop();      // 栈中有至少两个操作数，连续退栈
    switch (ch) {
        // 形成运算指令，并将结果入栈
        case '+': OPND.Push(X + Y); break;
        case '-': OPND.Push(X - Y); break;
        case '*': OPND.Push(X * Y); break;
        case '/': OPND.Push(X / Y); break;
        default: break;
    }
}
```

c) 当表达式的所有项都扫描并处理完后（即扫描到“#”），栈顶存放的就是最后的计算结果。

```
return OPND.getTop();
```

c) 运行结果（main.cpp）

i. 测试用中缀表达式：

```
char A[] = "1+2*(3-4)-5/6#";
char B[] = "2-3+4*5/(6+7)#";
```

ii. 分别转化为后缀表达式后进行计算，注释为预期的输出结果，实际运行结果与预期相符：

```
#include "SeqStack.cpp"
#include "Infix_to_Postfix.cpp"
#include "calculateRPN.cpp"

int main() {
    char A[] = "1+2*(3-4)-5/6#";
    char result[20] = {};
    InToPost(A, result);
    printf("%s 转化为后缀表达式为 %s\n", A, result);
    printf("运算结果为%f\n",calculateRPN(result));      // -1.833333

    char B[] = "2-3+4*5/(6+7)#";
    result[20] = {};
    InToPost(B, result);
    printf("%s 转化为后缀表达式为 %s\n", B, result);
    printf("运算结果为%f\n",calculateRPN(result));      // 0.538462
}
```

```
问题  终端  JUPYTER

clem@Connor:/mnt/c/Users/12879/Desktop/dataStructures$ cd ./projects/project02/version2usingcpp/
clem@Connor:/mnt/c/Users/12879/Desktop/dataStructures/projects/project02/version2usingcpp$ g++ main.cpp
clem@Connor:/mnt/c/Users/12879/Desktop/dataStructures/projects/project02/version2usingcpp$ ./a.out
1+2*(3-4)-5/6# 转化为后缀表达式为 1234-*+56/-#
运算结果为-1.833333
2-3+4*5/(6+7)# 转化为后缀表达式为 23-45*67+/+#
运算结果为0.538462
```

三、总结（实验中遇到的问题、取得的经验、感想等）

①实验中遇到了同时需要以 double 为数据类型和 char 为数据类型的两个栈（操作数栈和操作符栈），对于同一

个结构体通过两个 def 实现这一点是不行的，SelemType 只能是 double 或 char 其中之一的别名。如果定义两个结构体，则函数又需要都重写一遍，且不能重名。最后通过使用 c++的 class 和 template 解决了这一问题。

②实验中加深了对栈的把握和印象，对其应用有了更深刻的感受。它在往往用于数据的暂时储存。