# 计算器设计实验报告

## 一、需求分析

实验二

1、简单计算器。

请按照四则运算加、减、乘、除、幂（^）和括号的优先关系和惯例，编写计算器程序。要求：

1. 从键盘输入一个完整的表达式，以回车作为表达式输入结束的标志。
2. 输入表达式中的数值均为大于等于零的整数。中间的计算过程如果出现小数也只取整。

例如，输入：4+2\*5= 输出：14

输入：(4+2)\*(2-10)= 输出：-48

1. 由题可知，我们需要写出一个可以进行加减乘除和幂的计算器，这个计算器可以根据括号的优先关系和惯例来编写计算器程序。
2. 当我们输入一个完整的类似{3\*[4\*(5+6)+7]}↵的表达式时，计算器可以根据不同括号和数值的优先级来进行运算和操作，我们不妨以{3\*[4\*(5+6)+7]/9}^2↵当做计算表达式。

## 概要设计

在本实验中，我设计了六个基本函数、一个结构体和一个主函数。

分别为栈结构Stack、初始化栈函数void initStack(Stack \*stack, int maxSize)、入栈void push(Stack \*stack, char c)、出栈char pop(Stack \*stack)、判断括号是否配对bool isBalanced(char \*expression)、优先级比较函数int Priority(char a)、基础加减乘除运算函数int temporary\_result(char a,int b,int c)、还有main函数.

我在isBalanced中调用了Stack、initStack、push和pop。

在main函数中调用了isBalanced、Priority和temporary\_result。

## 详细设计

根据这个要求，首先我设计了两个关键点：

1. 程序错误时报错
2. 正常程序设计  
    根据问题一：由我们教学第五周的栈和队列练习—括号是否配对这一作业可以直接copy解决方法



我先阐释我关于这一代码要求的设计思路，首先我们创建栈的基本操作函数：栈结构、初始化栈、入栈、出栈；

在完成了这一系列基本函数的设置后，我开始进行isBalanced函数的编写，isBalanced函数旨在判定一个运算式的括号是否匹配，其基本思路如下：

先创建一个栈，然后遍历该运算式的字符数组，如果遇到‘(’‘[’‘{’，则将该字符入栈。如果遇到‘)’‘]’‘}’，则将字符出栈，并给currentChar赋值为该字符。并且执行出栈函数，将出栈的字符赋值为poppedChar

如果currentChar为‘)’而poppedChar不为‘）’，或者currentChar为‘}’poppedChar不为‘{’，或者currentChar为‘]’而poppedChar不为‘[’

则输出false，表示括号不匹配。

如果没有出现上述情况，则输出True，表示括号相匹配。

该部分代码如下：

*// 定义栈结构*

*typedef struct {*

*char \*data;*

*int size;*

*int top;*

*} Stack;*

*// 初始化栈*

*void initStack(Stack \*stack, int maxSize) {*

*stack->data = (char \*)malloc(sizeof(char) \* maxSize);*

*stack->size = maxSize;*

*stack->top = -1;*

*}*

*// 入栈*

*void push(Stack \*stack, char c) {*

*if (stack->top < stack->size - 1) {*

*stack->data[++stack->top] = c;*

*}*

*}*

*// 出栈*

*char pop(Stack \*stack) {*

*if (stack->top >= 0) {*

*return stack->data[stack->top--];*

*}*

*return '\0'; // 空字符表示栈已空*

*}*

*// 判断括号是否配对*

*bool isBalanced(char \*expression) {*

*Stack stack;*

*initStack(&stack, 100); // 选择足够大的栈容量*

*for (int i = 0; expression[i] != '\0'; i++) {*

*char currentChar = expression[i];*

*if (currentChar == '(' || currentChar == '[' || currentChar == '{') {*

*push(&stack, currentChar);*

*} else if (currentChar == ')' || currentChar == ']' || currentChar == '}') {*

*char poppedChar = pop(&stack);*

*if ((currentChar == ')' && poppedChar != '(') ||*

*(currentChar == ']' && poppedChar != '[') ||*

*(currentChar == '}' && poppedChar != '{')) {*

*free(stack.data);*

*return false; // 括号不匹配*

*}*

*}*

*}*

*// 所有括号都应该已经被匹配*

*free(stack.data);*

*return stack.top == -1;*

*}*

对于问题二：正常程序设计

首先我确立了两个基本函数，一为优先级比较函数，即确立了幂（^）的优先级是最高的，为3；乘（\*）和除（/）的优先级为2，加（+）和减（-）的优先级为1，中括号（[]）的优先级为0，大括号（{}）的优先级为-1。

其次我进行了基本算法结构设计，即当仅有一个符号a和两个int型整数b,c时，我们进行switch（a），然后根据运算符a的不同来进行算法的编写。两个函数的代码如下：

*//优先级比较函数*

*int Priority(char a){*

*switch (a) {*

*case '^':*

*return 3;*

*case '\*':*

*return 2;*

*case '/':*

*return 2;*

*case '+':*

*return 1;*

*case '-':*

*return 1;*

*case '[':*

*case ']':*

*return 0; // 中括号的优先级*

*case '{':*

*case '}':*

*return -1; // 大括号的优先级*

*default:*

*return 0;*

*}*

*}*

*//基础加减乘除运算函数*

*int temporary\_result(char a,int b,int c){*

*switch (a) {*

*case '+':*

*return b+c;*

*case '-':*

*return b-c;*

*case '\*':*

*return b\*c;*

*case '/':*

*return b/c;*

*case '^':*

*return pow(b,c);*

*default:*

*return 0;*

*}*

*}*

然后我们进行主函数的编写，首先定义两个字符数组char expression[100]和char na[100]，然后我们需要读入这个算式为字符数组，我们使用 fgets(expression, sizeof(expression), stdin);来进行算式的读入，再利用strcpy函数将expression的值赋值到na中，然后使用if (isBalanced(na))，来判断输入的算式是否出现括号不匹配的情况，如果出现了括号不匹配的情况那么就进行报错处理。

然后我们定义两个栈，分别是操作数栈和操作符栈：

// 操作数栈

int value[100];

int valueTop = -1;

// 操作符栈

char operatorStack[100];

int operatorTop = -1;

我们先遍历该字符数组，如果栈是数字，那么我们需要将数字存储到操作数栈中，但是由于我们的遍历是以字符为单位，而数字可能不止一个字符，即可能为多位的整数，此时即需要我们进行分析。

在这里，我利用了isdigit函数来进行问题的分析，我们先令operand=0，如果我们读入一个字符是数字字符，那么我们通过(expression[i] - '0')来得到该字符所代表的数值，并且令operand=operand\*10+(expression[i] - '0')来得到operand的值。

因为当读入第一个数字时，operand=0，那么operand即为(expression[i] - '0'),当读入第二个字符也为数字字符时，operand此时从各位转变为十位，即需要进行operand\*10后再与(expression[i] - '0')相加，后续多位数字的读入原理相同。

当我们读入的字符为括号时，如果为左括号‘(’‘[’‘{’，则直接入栈，operatorStack[++operatorTop] = expression[i];如果为右括号‘)’‘]’‘}’，则一直执行栈内的运算直到遇到相应的左括号。等到运算结束后将左括号弹出，然后继续进行下一个字符的读入。

运算函数规则如下：

当operatorTop 大于等于0 且 operatorStack[operatorTop] 不等于leftBracket时，我们令temp2=value[valueTop--]、temp1 = value[valueTop--]，temp\_operater= operatorStack[operatorTop--]即令temp2为操作数栈的最后一位数，temp1为操作数栈的倒数第二位数，temp\_operater为操作符栈的最后一个操作符。

然后调用函数temporary\_result(temp\_operater, temp1, temp2)得到运算结果，并赋值到value[++valueTop]上

当我们读入的字符为运算符时，我们可以执行栈内运算直到栈顶运算符优先级小于当前运算符，具体原理为：

如果operatorTop大于等于0且Priority函数得到的operatorStack[operatorTop]大于等于expression[i]，那么先执行operatorStack[operatorTop]所进行的运算，同样用上述运算函数规则进行。

最后执行剩余的运算，直到operatorTop=-1.

此时value[0]即为最终运算值。

main函数代码如下：

*int main() {*

*char expression[100];*

*char na[100];*

*fgets(expression, sizeof(expression), stdin);*

*strcpy(na,expression);*

*if (isBalanced(na)) {*

*printf("括号匹配，继续进行\n");*

*}*

*else {*

*printf("括号不配对,请重新输入\n");*

*exit(0);*

*}*

*// 操作数栈*

*int value[100];*

*int valueTop = -1;*

*// 操作符栈*

*char operatorStack[100];*

*int operatorTop = -1;*

*// 遍历表达式*

*for (int i = 0; expression[i] != '\0'; ++i) {*

*// 如果是数字，将数字转换并入栈*

*if (isdigit(expression[i])) {*

*int operand = 0;*

*while (isdigit(expression[i]))*

*{*

*operand = operand \* 10 + (expression[i] - '0');*

*++i;*

*}*

*--i; // 因为for循环中会自增*

*value[++valueTop] = operand;*

*}*

*else if (expression[i] == '(' || expression[i] == '[' || expression[i] == '{') {*

*// 如果是左括号，直接入栈*

*operatorStack[++operatorTop] = expression[i];*

*}*

*else if (expression[i] == ')' || expression[i] == ']' || expression[i] == '}') {*

*// 如果是右括号，执行栈内运算直到遇到对应的左括号*

*char leftBracket;*

*if (expression[i] == ')') leftBracket = '(';*

*else if (expression[i] == ']') leftBracket = '[';*

*else leftBracket = '{';*

*while (operatorTop >= 0 && operatorStack[operatorTop] != leftBracket)*

*{*

*int temp2 = value[valueTop--];*

*int temp1 = value[valueTop--];*

*char temp\_operater = operatorStack[operatorTop--];*

*value[++valueTop] = temporary\_result(temp\_operater, temp1, temp2);*

*}*

*// 弹出左括号*

*--operatorTop;*

*} else*

*{*

*// 如果是运算符，执行栈内运算直到栈顶运算符优先级小于当前运算符*

*while (operatorTop >= 0 && Priority(operatorStack[operatorTop]) >= Priority(expression[i])) {*

*int temp2 = value[valueTop--];*

*int temp1 = value[valueTop--];*

*char temp\_operater = operatorStack[operatorTop--];*

*value[++valueTop] = temporary\_result(temp\_operater, temp1, temp2);*

*}*

*// 当前运算符入栈*

*operatorStack[++operatorTop] = expression[i];*

*}*

*}*

*// 执行剩余的运算*

*while (operatorTop >= 0) {*

*int temp2 = value[valueTop--];*

*int temp1 = value[valueTop--];*

*char temp\_operater = operatorStack[operatorTop--];*

*value[++valueTop] = temporary\_result(temp\_operater, temp1, temp2);*

*}*

*// 计算表达式的值并输出结果*

*int result =value[0];*

*int len=strlen(expression);*

*expression[len-1]='\0';*

*printf("%s",expression);*

*printf(" = ");*

*printf("%d\n",result);*

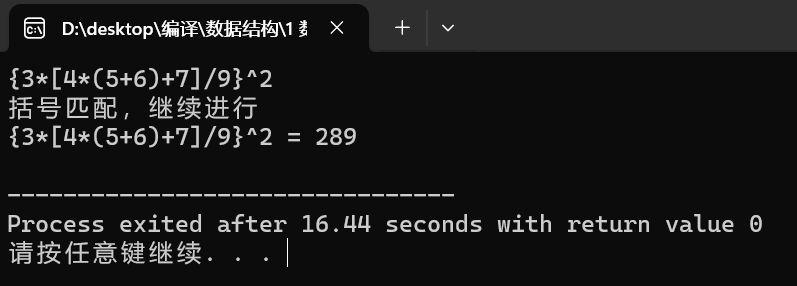
*return 0;*

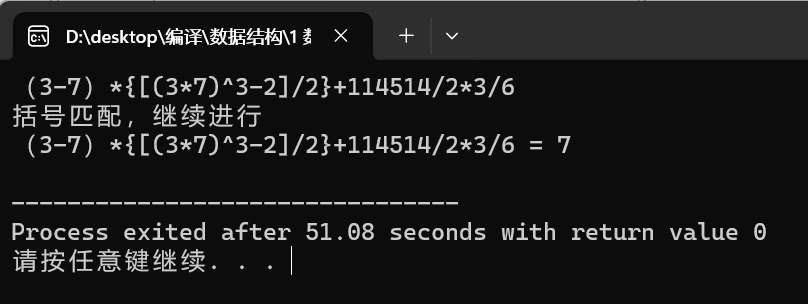
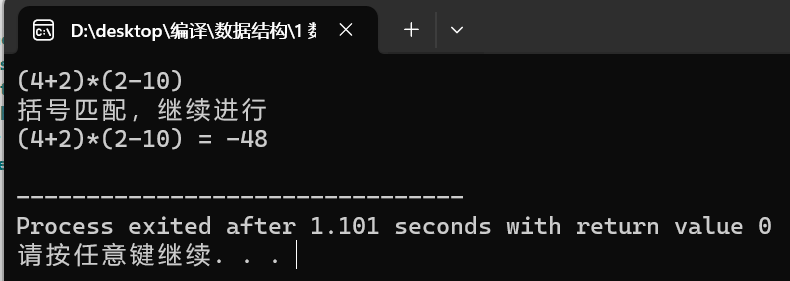
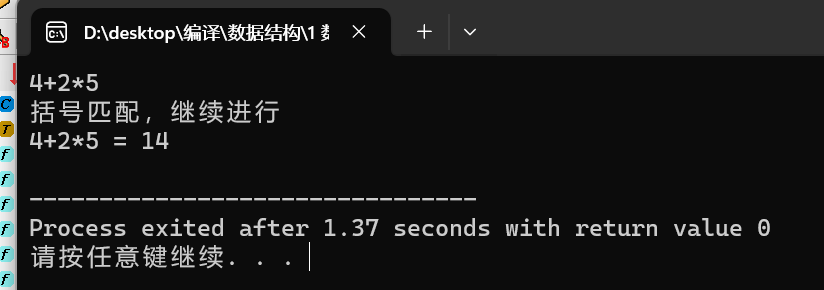
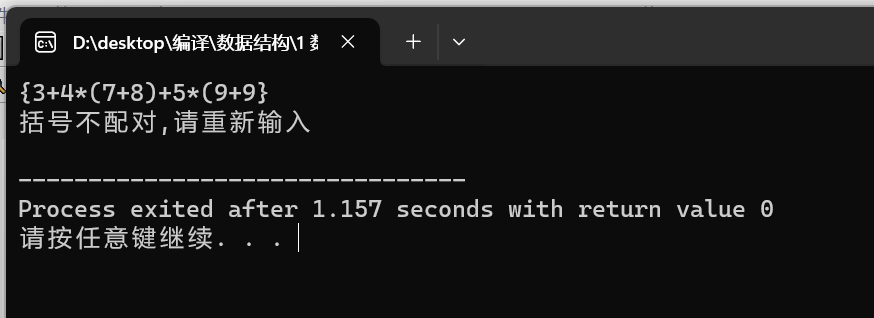
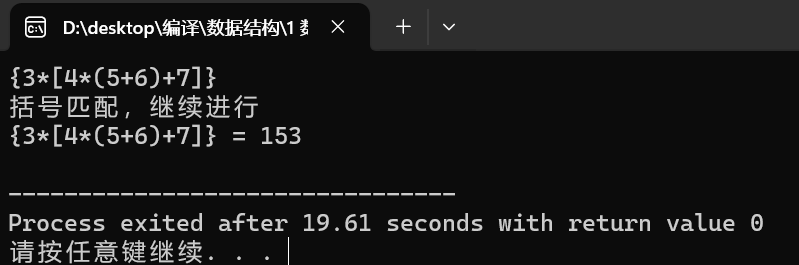
*}*

## 四、调试分析

在进行调试时，我也遇到了一些问题，首先为什么我要进行 char expression[100];char na[100];复制expression到na里，因为isBalanced函数里面的free函数会释放掉na的元素值，这也导致了最开始我没有用两个数组，而是直接用isBalanced(expression)时，后续运算发现无论输入什么都是零，最后通过调试后我发现了这个问题，所以利用双数组解决这个问题。

## 五、测试结果





## 六、附录

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

#include <ctype.h>

#include <stdbool.h>

// 定义栈结构

typedef struct {

char \*data;

int size;

int top;

} Stack;

// 初始化栈

void initStack(Stack \*stack, int maxSize) {

stack->data = (char \*)malloc(sizeof(char) \* maxSize);

stack->size = maxSize;

stack->top = -1;

}

// 入栈

void push(Stack \*stack, char c) {

if (stack->top < stack->size - 1) {

stack->data[++stack->top] = c;

}

}

// 出栈

char pop(Stack \*stack) {

if (stack->top >= 0) {

return stack->data[stack->top--];

}

return '\0'; // 空字符表示栈已空

}

// 判断括号是否配对

bool isBalanced(char \*expression) {

Stack stack;

initStack(&stack, 100); // 选择足够大的栈容量

for (int i = 0; expression[i] != '\0'; i++) {

char currentChar = expression[i];

if (currentChar == '(' || currentChar == '[' || currentChar == '{') {

push(&stack, currentChar);

} else if (currentChar == ')' || currentChar == ']' || currentChar == '}') {

char poppedChar = pop(&stack);

if ((currentChar == ')' && poppedChar != '(') ||

(currentChar == ']' && poppedChar != '[') ||

(currentChar == '}' && poppedChar != '{')) {

free(stack.data);

return false; // 括号不匹配

}

}

}

// 所有括号都应该已经被匹配

free(stack.data);

return stack.top == -1;

}

//优先级比较函数

int Priority(char a){

switch (a) {

case '^':

return 3;

case '\*':

return 2;

case '/':

return 2;

case '+':

return 1;

case '-':

return 1;

case '[':

case ']':

return 0; // 中括号的优先级

case '{':

case '}':

return -1; // 大括号的优先级

default:

return 0;

}

}

//基础加减乘除运算函数

int temporary\_result(char a,int b,int c){

switch (a) {

case '+':

return b+c;

case '-':

return b-c;

case '\*':

return b\*c;

case '/':

return b/c;

case '^':

return pow(b,c);

default:

return 0;

}

}

int main() {

char expression[100];

char na[100];

fgets(expression, sizeof(expression), stdin);

strcpy(na,expression);

if (isBalanced(na)) {

printf("括号匹配，继续进行\n");

}

else {

printf("括号不配对,请重新输入\n");

exit(0);

}

// 操作数栈

int value[100];

int valueTop = -1;

// 操作符栈

char operatorStack[100];

int operatorTop = -1;

// 遍历表达式

for (int i = 0; expression[i] != '\0'; ++i) {

// 如果是数字，将数字转换并入栈

if (isdigit(expression[i])) {

int operand = 0;

while (isdigit(expression[i]))

{

operand = operand \* 10 + (expression[i] - '0');

++i;

}

--i; // 因为for循环中会自增

value[++valueTop] = operand;

}

else if (expression[i] == '(' || expression[i] == '[' || expression[i] == '{') {

// 如果是左括号，直接入栈

operatorStack[++operatorTop] = expression[i];

}

else if (expression[i] == ')' || expression[i] == ']' || expression[i] == '}') {

// 如果是右括号，执行栈内运算直到遇到对应的左括号

char leftBracket;

if (expression[i] == ')') leftBracket = '(';

else if (expression[i] == ']') leftBracket = '[';

else leftBracket = '{';

while (operatorTop >= 0 && operatorStack[operatorTop] != leftBracket)

{

int temp2 = value[valueTop--];

int temp1 = value[valueTop--];

char temp\_operater = operatorStack[operatorTop--];

value[++valueTop] = temporary\_result(temp\_operater, temp1, temp2);

}

// 弹出左括号

--operatorTop;

} else

{

// 如果是运算符，执行栈内运算直到栈顶运算符优先级小于当前运算符

while (operatorTop >= 0 && Priority(operatorStack[operatorTop]) >= Priority(expression[i])) {

int temp2 = value[valueTop--];

int temp1 = value[valueTop--];

char temp\_operater = operatorStack[operatorTop--];

value[++valueTop] = temporary\_result(temp\_operater, temp1, temp2);

}

// 当前运算符入栈

operatorStack[++operatorTop] = expression[i];

}

}

// 执行剩余的运算

while (operatorTop >= 0) {

int temp2 = value[valueTop--];

int temp1 = value[valueTop--];

char temp\_operater = operatorStack[operatorTop--];

value[++valueTop] = temporary\_result(temp\_operater, temp1, temp2);

}

// 计算表达式的值并输出结果

int result =value[0];

int len=strlen(expression);

expression[len-1]='\0';

printf("%s",expression);

printf(" = ");

printf("%d\n",result);

return 0;

}