

数据结构课程设计  
项目说明文档

# 算数表达式求值

作者姓名:	<u>汪明杰</u>
学 号:	<u>1851055</u>
指导教师:	<u>张 颖</u>
学院专业:	<u>软件学院 软件工程</u>



同济大学  
Tongji University

# 目录

1 项目分析 .....	4
1.1 项目背景 .....	4
1.2 项目需求分析 .....	4
1.3 项目要求 .....	5
1.3.1 功能要求 .....	5
1.3.2 输入格式 .....	5
1.3.3 输出格式 .....	5
1.3.4 项目示例 .....	5
2 项目设计 .....	6
2.1 数据结构设计 .....	6
2.2 类设计 .....	6
2.2.1 结点类 (ListNode) .....	7
2.2.2 双向链表类 (List) .....	7
2.2.3 栈类 (Stack) .....	8
3 项目实施 .....	10
3.1 运算符设计 .....	11
3.2 运算符处理 .....	11
3.3 运算符计算 .....	13
3.4 栈的操作 .....	14
3.5 测试函数 .....	15
4 项目测试 .....	17
4.1 正整数加法测试 .....	17
4.2 正整数减法测试 .....	17
4.3 带负数的整数加法测试 .....	18
4.4 带负数的整数减法测试 .....	18
4.5 小数加法测试 .....	19
4.6 小数减法测试 .....	19
4.7 整数乘法运算 .....	20
4.8 小数乘法测试 .....	20
4.9 整数除法测试 .....	21

4.10 小数除法测试.....	21
4.11 幂次运算测试.....	22
4.12 模运算测试.....	22
4.13 混合运算测试.....	23
4.16 带括号的运算测试 .....	23
4.17 不带等号的运算测试.....	24
4.18 带括号的混合运算 .....	24
4.19 24个样例测试 .....	25

# 1 项目分析

## 1.1 项目背景

计算自古以来就是在人类发展中极为重要的一环，算力的大小很大层次上反映了科学水平进步的程度。在远古时期，人们最初是用手指进行计算。用手指计算虽然很方便，但计算范围有限，计算结果也无法储存。于是人们开始使用绳子、石子等工具来增加计算能力，正也正是“上古结绳而治”。

随着历史的发展，算盘、计算器、计算机等逐渐出现在历史的舞台上。虽然现在我们可以在计算机上进行各种操作，但是计算仍然是各种高级操作最基本的组成。计算自然而然，也是我们急需解决的一个问题。

本项目正是基于现实生活中对于计算的需求而产生的。

## 1.2 项目需求分析

针对于算术表达式求值问题，本项目在设计过程中，应当考虑到满足以下需求：

- ✓ **功能完善**

系统需要满足所要求的各种运算，包括：加减、乘除取余、乘方和括号等运算符，同时也需要能够处理包括正负在内的单目运算符。

- ✓ **执行效率高**

针对表达式相对比较复杂的情况，本系统也应该具有在较短时间内求解出正确答案的能力。

- ✓ **健壮性**

当用户输入的数据非法时，系统应当能够识别并处理错误，而非直接崩溃退出。

- ✓ **友好型**

系统应当给予合理的引导提示，以便于用户使用。同时，提供一些默认样例，便于用户验证计算器功能正确性。

## 1.3 项目要求

### 1.3.1 功能要求

从键盘上输入中缀算数表达式，包括括号，计算出表达式的值。

程序对所有输入的表达式作简单的判断，如表达式有错，能给出适当的提示。

程序需支持：包括加减，乘除取余，乘方和括号等操作符（优先级：等于<括号<加减<乘除取余<乘方）；单目运算符：+或-。

### 1.3.2 输入格式

中缀算数表达式，包括括号和上述运算符。

### 1.3.3 输出格式

表达式的值。若表达式有错，则给出提示。

### 1.3.4 项目示例

序号	输入	输出	说明
1	1+2-3=	0	基本加减法运算
2	3*6/2=	9	基本乘除法运算
3	3^2%11=	9	幂次和求模运算
4	(2-3)/2+3=	2.5	带括号的运算
5	-2*(3+5)+2^3/4=	-14	混合运算
6	3/0=	除数不能为 0	表达式有错的情况

## 2 项目设计

### 2.1 数据结构设计

如上述功能分析所示，本项目实际要求解完成一个中缀表达式的求值问题。针对于中缀表达式的求解，主要有两种方法：第一种是先将中缀表达式转化为后缀表达式，即逆波兰表达式，然后就是后缀表达式的求解问题；第二种则是直接求解。

本项目中采用第二种方法求解，主要思路是：首先建立两个栈，分别是运算符栈和数字栈。接下来，开始对表达式进行读取，如果读取到数字则直接压入数字栈中；否则则查看当前运算符栈中是否存在内容。如果存在内容且当前运算符优先级较高，则将当前运算符压入运算符栈；否则从操作数栈中弹出一个或两个运算数进行运算再压回操作数栈。

本系统各种操作的时间复杂度如下：

- ✓ 计算表达式结果： $O(n)$

### 2.2 类设计

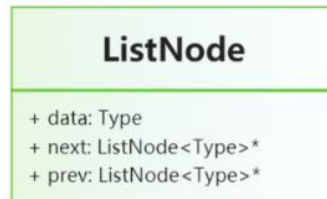
需要注意的是，本项目中使用到了栈（Stack）这一数据结构，而栈的底层数据类型则采用了链表进行实现。经典的链表一般包括两个抽象数据类型（ADT）——链表结点类（ListNode）与链表类（List），而两个类之间的耦合关系可以采用嵌套、继承等多种关系。

为了实现代码的复用性，本系统实现了一个**链表**。采用 struct 描述链表结点类（ListNode），这样使得链表结点类（List）可以直接访问链表结点而不需要定义友元关系。本系统实现的链表结构各种操作的时间复杂度如下：

- ✓ 插入操作： $O(n)$
- ✓ 删除操作： $O(n)$
- ✓ 查询操作： $O(n)$
- ✓ 遍历操作： $O(n)$

### 2.2.1 结点类 (ListNode)

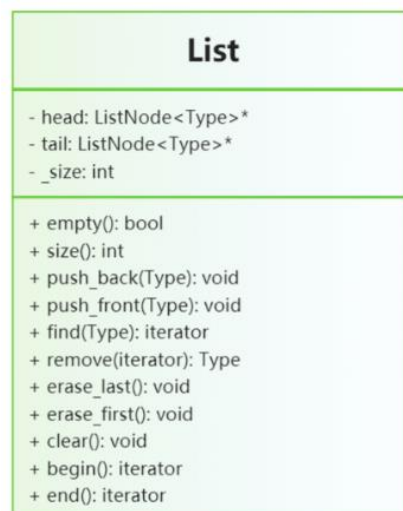
链表的结点中存储的数据有链表数据、后继结点和前驱结点。其 UML 图如下所示：



### 2.2.2 双向链表类 (List)

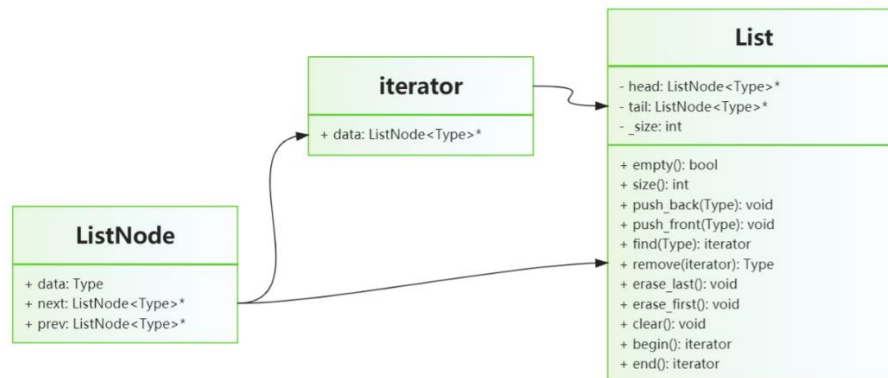
链表的实现原理大同小异，不同之处在于：是否带头结点、是否带尾结点、每一个结点是否带前驱结点等。为了使得链表中各种操作的时间复杂度都尽可能低，因此这里选择了带头结点和尾结点的双向链表来实现链表中的各种操作。也即：选择了牺牲空间来达到降低时间复杂度的效果。

链表的 UML 图如下所示：



为了便于对链表进行遍历、插入、删除、查找等操作，增加了一个 `iterator` 类。`iterator` 类内部存储一个链表节点指针，同时，通过运算符重载相应的自增、自减、判等等操作。

`iterator` 类、`ListNode` 类和 `List` 类的关系如下图所示：



其中，List 类中的主要函数如下所示：

◆ **inline int size()const**

返回链表中结点的个数，不包括头结点。

◆ **inline bool empty()const**

判断链表是否为空，也即链表中结点的个数是否为 0。

◆ **void push\_back(Type data)**

在链表尾部插入新的数据，也即新增一个结点并且加入到链表末端。

◆ **void push\_front(Type data)**

在链表头部插入新的数据，也即新增一个结点并且加入到链表的头部。

◆ **iterator find(const Type& data)const**

在链表中查找值为 data 的元素是否存在，返回该位置的迭代器，若查找失败返回空指针对应的迭代器。

◆ **Type remove(iterator index)**

移除迭代器所处位置的元素，返回移除位置元素的值。

◆ **void erase\_last()**

移除链表末端的元素，即最后的结点。

◆ **void erase\_first()**

移除链表首端的元素，即第一个结点。

◆ **void clear()**

清空链表，即删除链表中所有的结点。

◆ **iterator begin()**

返回链表第一个元素所在位置的迭代器。

◆ **iterator end()**

返回链表尾结点后的空结点的迭代器。

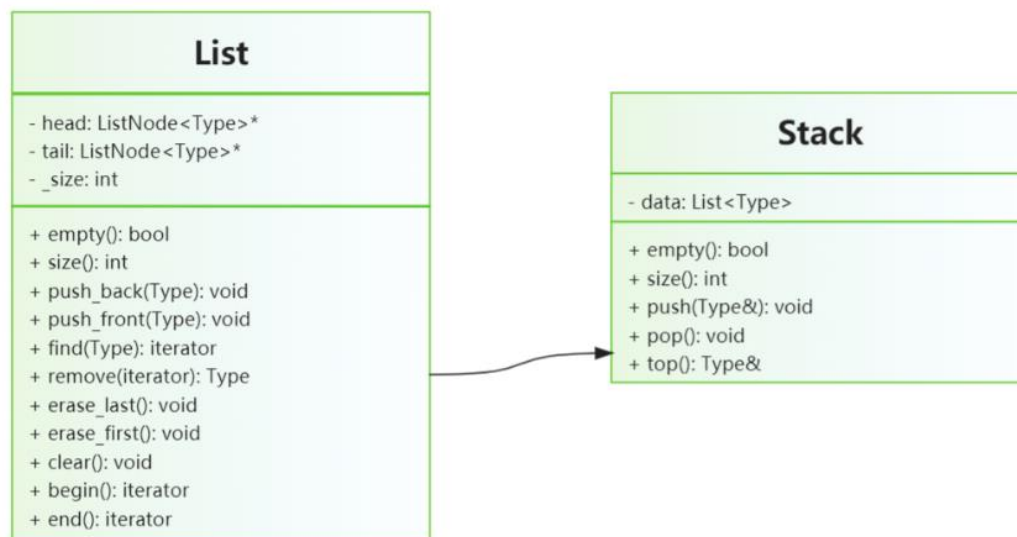
### 2.2.3 栈类（Stack）

栈（Stack）又名堆栈，它是一种运算受限的线性表。限定仅在表尾进行插入和删除操作的线性表。这一端被称为栈顶，相对地，把另一端称为栈底。向一个



栈插入新元素又称作进栈、入栈或压栈，它是把新元素放到栈顶元素的上面，使之成为新的栈顶元素；从一个栈删除元素又称作出栈或退栈，它是把栈顶元素删除掉，使其相邻的元素成为新的栈顶元素。因此，栈是一种先进后出（First In Last Out）的数据结构。

本项目中的栈使用链表作为底层数据结构，其 UML 图如下所示：



正如下图所示，栈中主要的操作函数包括了：

◆ `inline bool empty()const`

判断栈是否为空，也即栈内部的链表节点个数是否为 0。

◆ `inline int size()const`

返回栈的大小，也即栈内部链表的节点个数。

◆ `void push(const Type& i)`

在栈尾部加入一个元素，也即入栈。

◆ `void pop()`

取栈中最后一个元素，也即出栈。

◆ `const Type& top()const`

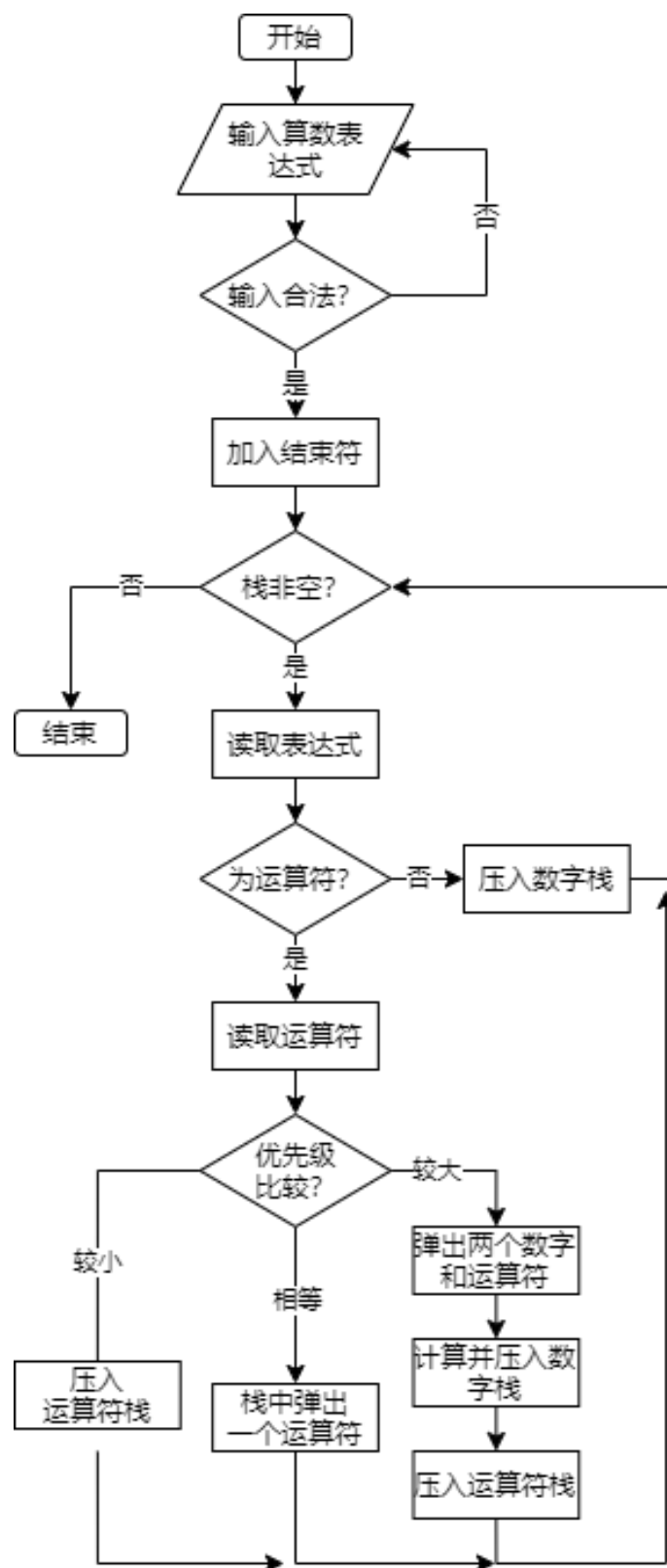
获取栈顶的元素值。

通过上述函数，即实现了栈最基本的操作。其中，栈的各个操作的时间复杂度如下所示：

- ✓ 入栈操作：O(1)
- ✓ 出栈操作：O(1)
- ✓ 读取操作：O(1)

### 3 项目实施

本项目中在实现过程中，主要流程图如下所示：



## 3.1 运算符设计

根据项目要求，本项目需要实现 11 种运算符的计算。因此自然而然地，设计 11 种运算符也是相当重要的。为了便于运算符的表示，项目先用 `enum` 将各种运算符表示出来，如下所示：

- ✓ 等号 (EQU) : +
- ✓ 正号 (POS) : +
- ✓ 负号 (NEG) : -
- ✓ 相加 (ADD) : +
- ✓ 相减 (SUB) : -
- ✓ 相乘 (MUL) : \*
- ✓ 相除 (DIV) : /
- ✓ 幂次 (POW) : ^
- ✓ 取模 (MOD) : %
- ✓ 左括号 (LP) : (
- ✓ 右括号 (RP) : )

同时，由于在运算符栈的实现过程中需要考虑各个运算符的优先级。因此，本项目中定义了一个优先级表，便于直接查询任意两个运算符之间的优先级，如下所示：

```
static const char order[11][11] = {
    '=', '<', 'x', '<', '<', '<', '<', '<', '<', '<', '<',
    'x', '<', '=', '<', '<', '<', '<', '<', '<', '<', '<',
    '>', '?', '>', 'x', 'x', '>', '>', '>', '>', '>', '>',
    '>', '<', '>', '<', '<', '>', '>', '>', '>', '>', '>',
    '>', '<', '>', '<', '<', '>', '>', '>', '>', '>', '>',
    '>', '<', '>', '<', '<', '>', '>', '<', '<', '<', '<',
    '>', '<', '>', '<', '<', '>', '>', '<', '<', '<', '<',
    '>', '<', '>', '<', '<', '>', '>', '>', '>', '<', '>',
    '>', '<', '>', '<', '<', '>', '>', '>', '>', '<', '>',
    '>', '<', '>', '<', '<', '>', '>', '>', '>', '>', '>',
    '>', '<', '>', '<', '<', '>', '>', '>', '>', '<', '>'
};
```

## 3.2 运算符处理

当读取字符串时，如果读取到了字符，则应该判断该运算符为何种类型。因此，本项目通过 `dispatch` 函数实现了对运算符的识别，函数内容如下所示：

```

OPERATORS dispatch(char op, int lastObj)
{
    switch (op)
    {
        case '+':
            return ((lastObj == -1 || lastObj == RP) ? ADD : POS);
        case '-':
            return ((lastObj == -1 || lastObj == RP) ? SUB : NEG);
        case '*':
            return MUL;
        case '/':
            return DIV;
        case '^':
            return POW;
        case '%':
            return MOD;
        case '=':
        case '\\0':
            return EQU;
        case '(':
            if (lastObj == RP)
            {
                printf("\\n 与上一个右括号间没有运算符");
                exit(-1);
            }
            return LP;
        case ')':
            if (lastObj == LP)
            {
                printf("\\n 两个括号里没有内容");
                exit(-1);
            }
            return RP;
        default:
            printf("\\n 尚不支持该运算符! ");
            exit(-1);
    }
}

```

### 3.3 运算符计算

如项目设计中所述，如果压入运算符优先级低于已存在运算符优先级，则需要从操作数栈中弹出一个或两个运算数进行运算再压回操作数栈。在本项目中，该操作通过 `calculate` 函数实现，其具体内容如下所示：

```
double calculate(OPERATORS op, double op1, double op2)
{
    switch (op)
    {
        case POS:
            return op1;
        case NEG:
            // 负数
            return -op1;
        case ADD:
            // 相加
            return op1 + op2;
        case SUB:
            // 相减
            return op1 - op2;
        case MUL:
            // 相乘
            return op1 * op2;
        case POW:
            // 幂次
            return pow(op1, op2);
        case DIV:
            if (op2 == 0)
            {
                printf("\n 除数不能为 0");
                exit(-1);
            }
            return op1 / op2;
        case MOD:
            if (op2 == 0)
            {
                printf("\n 除数不能为 0");
                exit(-1);
            }
            return fmod(op1, op2);
    }
    return 0;
}
```

## 3.4 栈的操作

本项目对表达式进行读取的过程如下所示：

如果读取到数字，则直接压入数字栈中。

如果读取到的是运算符，则查看当前运算符栈中是否存在内容。如果存在内容且当前运算符优先级较高，则将当前运算符压入运算符栈；否则从操作数栈中弹出一个或两个运算数，进行运算（即 3.3 操作）再压回操作数栈。

为了实现上述操作，本项目通过 `dealWith` 函数实现了该功能，如下所示：

```
double dealWith(const char *s)
{
    Stack<double> opnd;
    Stack<OPERATORS> optr;

    // 添加等号运算符
    int lastObj = EQU;
    optr.push(EQU);

    // 递归判断
    while (!optr.empty())
    {
        while (isblank(*s))
        {
            s++;
        }
        if (isdigit(*s))
        {
            // 如果是数字
            double x; int len;
            sscanf(s, "%lf%n", &x, &len);
            opnd.push(x);
            s += len;
            lastObj = -1;
        }
        else
        {
            // 如果是操作符
            OPERATORS newOp = dispatch(*s, lastObj);
            switch (order[optr.top()][newOp])
            {
                case '<':
                    optr.push(newOp); s++;
                    lastObj = newOp; break;
            }
        }
    }
}
```

```

        case '=': //退栈
            optr.pop(); s++;
            lastObj = newOp; break;
        case '>':
        {
            OPERATORS op = optr.top(); optr.pop();
            if (op == POS || op == NEG)
            {
                double opnd1 = opnd.top(); opnd.pop();
                opnd.push(calculate(op, opnd1, 0));
            }
            else
            {
                double opnd2 = opnd.top(); opnd.pop();
                double opnd1 = opnd.top(); opnd.pop();
                opnd.push(calculate(op, opnd1, opnd2));
            }
            break;
        }
        default:
            cerr << "请检查表达式是否有误" << endl;
            exit(-1);
    }
}
return opnd.top();
}

```

### 3.5 测试函数

此外，本系统为了方便用户检查默认的计算是否有误。因此，提供了测试函数，从而便于打印。测试函数内容如下所示：

```

void test(const char* str, double correctAnswer)
{
    static int index = 1;
    double calculateAnswer = dealWith(str);
    printf("-----\n");
    printf("%2d    %-10s    \t", index, str);
    cout << calculateAnswer;
    printf("          \t");
    cout << correctAnswer;
}

```

```
printf("          \t%s\n", ((correctAnswer - calculateAnswer) < 1e-6  
? "T" : "F"));  
    ++index;  
}
```

通过测试函数以及提供的默认 24 个测试用例，即可验证本系统的功能相对比较完善。



## 4 项目测试

### 4.1 正整数加法测试

测试数据：1+2=

预期结果：3

实验结果：



### 4.2 正整数减法测试

测试数据：6-3=

预期结果：3

实验结果：



### 4.3 带负数的整数加法测试

测试数据:  $-6+2=$

预期结果:  $-4$

实验结果:



### 4.4 带负数的整数减法测试

测试数据:  $-3--1=$

预期结果:  $-2$

实验结果:



## 4.5 小数加法测试

测试数据：6.3+2.7=

预期结果：9

实验结果：



```
D:\学习文件\数据结构\数据结构课设\4-算术表达式求值\Debug\4-算术表达式求值.exe
现在请手动输入表达式：6.3+2.7=
计算结果：9
```

## 4.6 小数减法测试

测试数据：6.7-2.8=

预期结果：3.9

实验结果：



```
D:\学习文件\数据结构\数据结构课设\4-算术表达式求值\Debug\4-算术表达式求值.exe
现在请手动输入表达式：6.7-2.8=
计算结果：3.9
```

## 4.7 整数乘法运算

测试数据：27\*68=

预期结果：1836

实验结果：



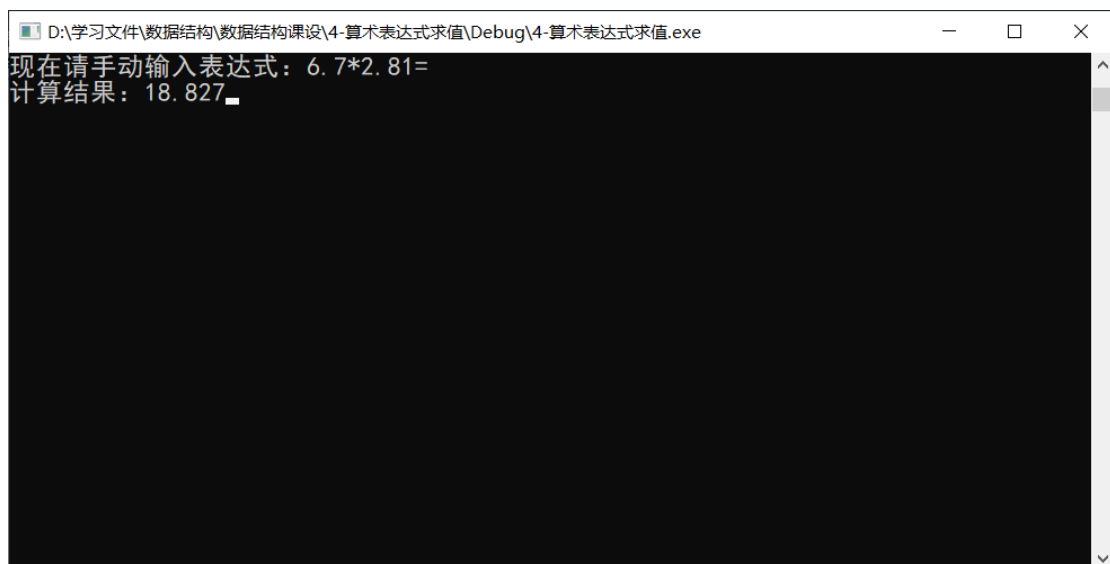
```
D:\学习文件\数据结构\数据结构课设\4-算术表达式求值\Debug\4-算术表达式求值.exe
现在请手动输入表达式：27*68=
计算结果：1836_
```

## 4.8 小数乘法测试

测试数据：6.7\*2.81=

预期结果：18.827

实验结果：



```
D:\学习文件\数据结构\数据结构课设\4-算术表达式求值\Debug\4-算术表达式求值.exe
现在请手动输入表达式：6.7*2.81=
计算结果：18.827_
```

## 4.9 整数除法测试

测试数据：768/3=

预期结果：256

实验结果：



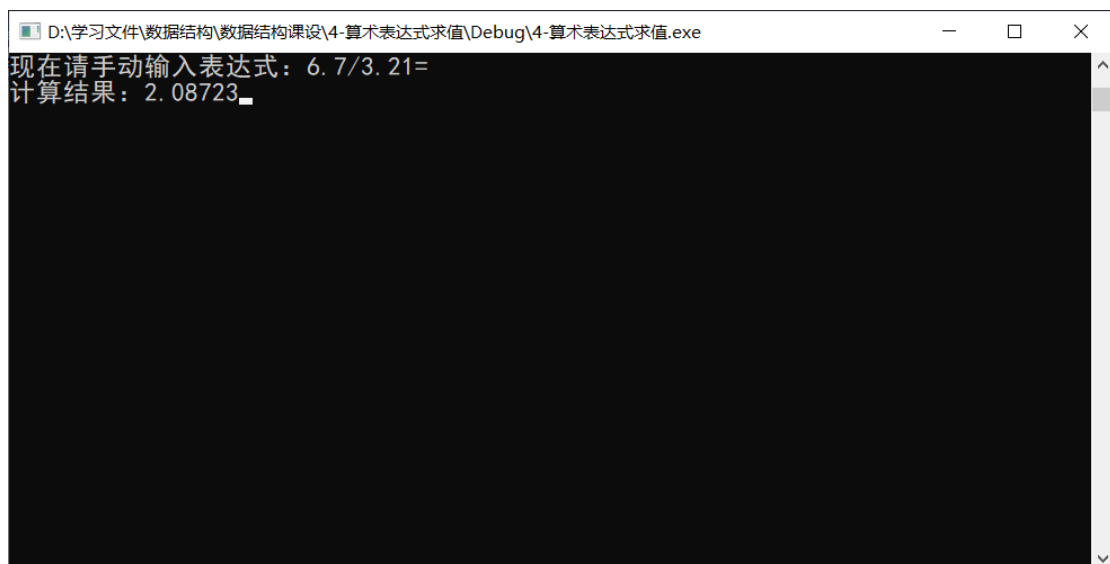
```
D:\学习文件\数据结构\数据结构课设\4-算术表达式求值\Debug\4-算术表达式求值.exe
现在请手动输入表达式：768/3=
计算结果：256_
```

## 4.10 小数除法测试

测试数据：6.7/3.21=

预期结果：2.087

实验结果：



```
D:\学习文件\数据结构\数据结构课设\4-算术表达式求值\Debug\4-算术表达式求值.exe
现在请手动输入表达式：6.7/3.21=
计算结果：2.08723_
```

## 4.11 幂次运算测试

测试数据： $6^3=$

预期结果：216

实验结果：



## 4.12 模运算测试

测试数据： $21\%4=$

预期结果：1

实验结果：

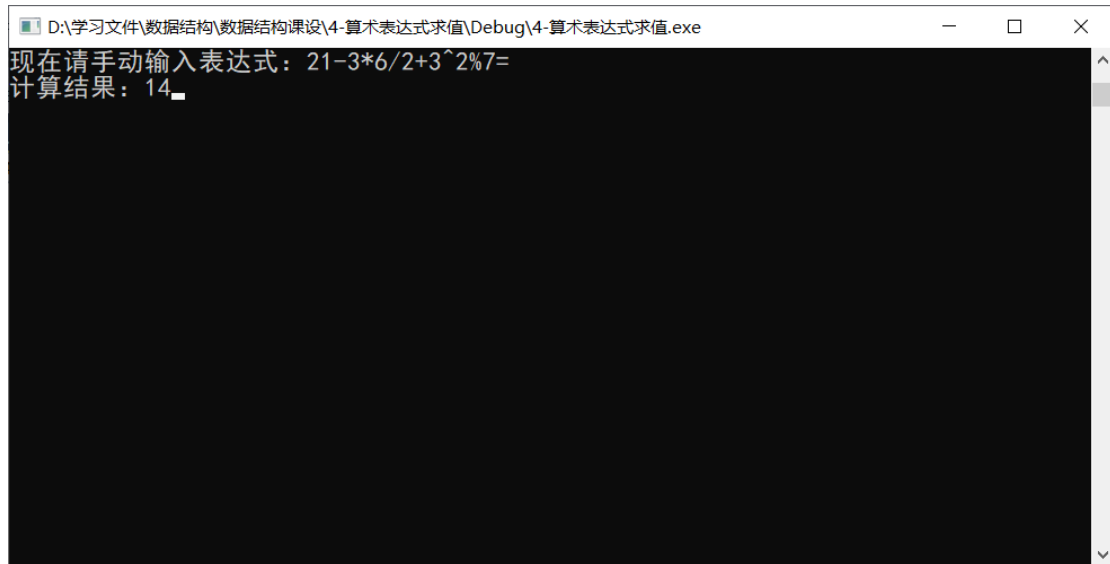


### 4.13 混合运算测试

测试数据： $21-3*6/2+3^2\%7=$

预期结果：14

实验结果：

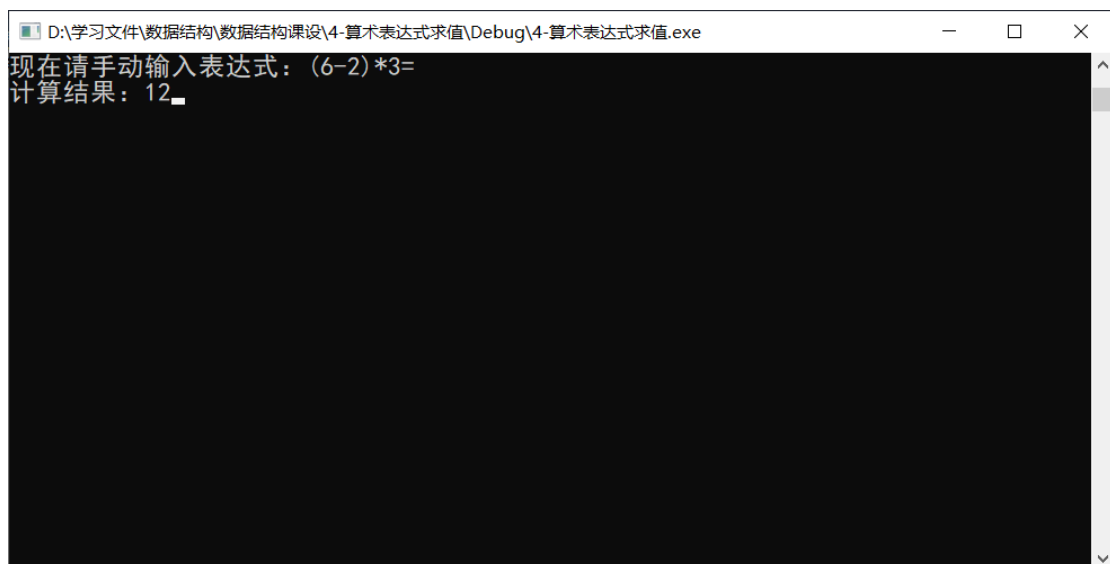


### 4.16 带括号的运算测试

测试数据： $(6-2)*3=$

预期结果：12

实验结果：



## 4.17 不带等号的运算测试

测试数据：21/7+3%2

预期结果：

实验结果：



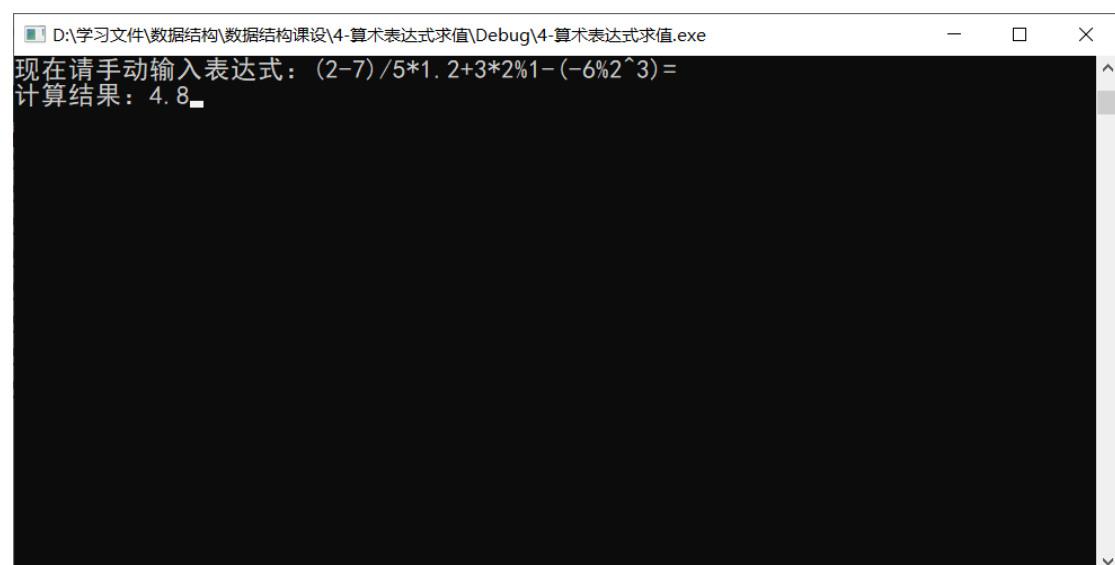
```
D:\学习文件\数据结构\数据结构课设\4-算术表达式求值\Debug\4-算术表达式求值.exe
现在请手动输入表达式：21/7+3%2
计算结果：4_
```

## 4.18 带括号的混合运算

测试数据：(2-7)/5\*1.2+3\*2%1-(-6%2^3)=

预期结果：4.8

实验结果：



```
D:\学习文件\数据结构\数据结构课设\4-算术表达式求值\Debug\4-算术表达式求值.exe
现在请手动输入表达式：(2-7)/5*1.2+3*2%1-(-6%2^3)=
计算结果：4.8_
```



## 4.19 24 个样例测试

测试数据：无

预期结果：24 个样例正确答案

实验结果：

序号	表达式	计算答案	正确答案	正确性
1	$1+2=$	3	3	T
2	$1.9+3.1=$	5	5	T
3	$6-3=$	3	3	T
4	$2-4=$	-2	-2	T
5	$-0.3+2.2=$	1.9	1.9	T
6	$1.6-2.7=$	-1.1	-1.1	T
7	$2*10=$	20	20	T
8	$-6*3=$	-18	-18	T
9	$0.6*0.1=$	0.06	0.06	T
10	$4/4=$	1	1	T
11	$-8/2=$	-4	-4	T
12	$2/4=$	0.5	0.5	T
13	$0.78/0.2=$	3.9	3.9	T
14	$2^3=$	8	8	T
15	$0.6^2=$	0.36	0.36	T
16	$6\%2=$	0	0	T
17	$7\%6=$	1	1	T
18	$21\%5=$	1	1	T
19	$(2+1)*3=$	9	9	T
20	$1-(2+3)=$	-4	-4	T
21	$2*(1+3/(6-5))=$	8	8	T
22	$(6-4)\%4=$	2	2	T
23	$6-2*3$	0	0	T
24	$10-(2+3)/2$	7.5	7.5	T

24个样例测试完毕