# 《数据结构与算法分析》

# 实验报告

| | |
|---|---|
| 实验名称 | 基于栈的中缀算术表达式求值 |
| 姓名 | 郑罡 |
| 学号 | 23060826 |
| 院系 | 自动化（**人工智能**）**学院** |
| 专业 | **人工智能** |
| 班级 | 23061011 |
| 实验时间 | 2024.4.19 |

# 一、实验目的

1. 掌握栈的基本操作算法的实现，包括栈初始化、进栈、出栈、取栈顶元素等。
2. 掌握利用栈实现中缀表达式求值的算法。

# 二、实验设备与环境

Windows 11 (64-bit), VSCode + g++ (CPP17 标准)

# 三、实验内容与结果

实验内容与结果较长共包含 6 页内容，将附在本页后。

# 四、实验总结及体会

通过本次实验，笔者成功利用栈数据结构实现了表达式求值的功能。该程序能够正确计算包含二元运算符、括号和浮点数的算术表达式。笔者分别采用了栈来存储操作数和操作符，并根据操作符优先级合理地对栈进行操作，从而得到最终的计算结果。

不过，该程序在处理某些异常输入时表现还有待改进，如空输入、多余运算符和括号等情况下的行为需要进一步优化。另外，该程序目前只支持二元运算符，对于其他运算符的处理也可以作为后续的拓展方向。

此外，本实验的实验结果并非完全符合原题目要求，但笔者认为每读入一行输出一次结果的交互方式更符合直觉，因此采用此方式。如需完全实现原题要求，只需将结果保存至一个 vector 中即可。

本部分所有与课程实验相关的代码都将放置在Github https://github.com/Jerry050512/Data-Structure-Experiment-Tasks上。原来的仓库将被删除并迁移至新仓库。

## 任务要求

利用栈实现表达式求值，其中只需要考虑二元运算符与括号，数字均假定为浮点数。

## 设计思路

在之前的课程中，已经在 `stack.cpp` 中实现了栈的类，可以很方便的使用。也实现过个位整数的四则运算。

只需要简单的修改就可以实现浮点数的求值。

## 栈的设计

在这次的实验中需要用的栈的以下功能：

```
template <typename T, size_t MAX_SIZE = 100>
class Stack
{
private:
    T data[MAX_SIZE];
    size_t top_index;

public:
    Stack() : top_index(0) {}

    // 判断栈是否为空
    bool isEmpty() const{}

    // 判断栈是否已满
    bool full() const{}

    // 压入栈
    void push(const T &value) {}

    // 弹出栈
    T &pop() {}

    // 获取栈顶元素进行操作
    T &top() {}
};
```

这里就涉及到最终代码的一个缺陷，就是 `push()` 方法会检查栈是否已满，我们的代码没有处理如果计算遇到栈满的情况。

## 求值算法

基础算法其实很简单，遵循以下步骤：

1. 构建优先级表；
2. 遍历表达式，遇到数字则压入栈中；
3. 遇到左括号，则压入栈中；
4. 遇到右括号，则弹出栈中元素，直到遇到左括号，此时弹出的元素为二元运算符，将栈顶两个元素弹出，计算结果压入栈中；
5. 遇到二元运算符，则弹出栈中元素，直到遇到比当前运算符优先级高的运算符，此时弹出的元素为二元运算符，将栈顶两个元素弹出，计算结果压入栈中；

这是原来针对于整数计算的代码：

```
/**
 * 计算给定表达式的值。
 *
 * @param expression 要计算的表达式字符串。
 * @return 表达式的计算结果。
 */
int evaluateExpression(string expression)
{
    Stack<int> values; // 用于存储操作数的栈
    Stack<char> ops;   // 用于存储操作符的栈

    for (char c : expression)
    {
        if (isdigit(c))
        {
            values.push(c - '0'); // 将操作数压入值栈
        }
        else if (c == '(')
        {
            ops.push(c); // 将开括号压入操作符栈
        }
        else if (c == ')')
        {
            // 计算括号内的子表达式
            while (!ops.isEmpty() && ops.top() != '(')
            {
                int b = values.pop();
                int a = values.pop();
                char op = ops.pop();
                values.push(applyOperation(a, b, op));
            }
            if (!ops.isEmpty())
            {
                ops.pop(); // 从操作符栈中弹出开括号
            }
        }
        else if (opPrecedence.count(c))
        {
```

```
            // 处理操作符
            while (!ops.isEmpty() && opPrecedence.count(ops.top()) &&
opPrecedence[ops.top()] >= opPrecedence[c])
            {
                int b = values.pop();
                int a = values.pop();
                char op = ops.pop();
                values.push(applyOperation(a, b, op));
            }
            ops.push(c); // 将当前操作符压入操作符栈
        }
    }

    // 处理剩余的操作符
    while (!ops.isEmpty())
    {
        int b = values.pop();
        int a = values.pop();
        char op = ops.pop();
        values.push(applyOperation(a, b, op));
    }

    return values.pop(); // 最终结果位于值栈的顶部
}
```

我们只需小作修改，就可以实现浮点数的求值。

### 对浮点数的处理

为了引入对浮点数的处理，首先我们需要将栈`values`，变量`a` `b`和函数`applyOperation()`都修改为`double`类型。

接下来引入两个变量，`prev_was_digit`和`multiple`，一个用于记录上一个字符是否为数字，一个用于记录下一个拼接的数字行为。

首先我们在`evaluate_expression()`函数初始化两个变量：

```
bool prev_was_digit = false;
double multiple = 10;
```

然后在判断当前字符是否为数字的分支时，进行如下操作：

```
if(prev_was_digit)
{
    // 如果是数字，则将数字拼接起来
    if(multiple > 1)
    {
        // 整数部分的拼接
        values.top() = values.top() * multiple + (c - '0');
    }
    else
    {
        // 小数部分的拼接
        values.top() = values.top() + (c - '0') * multiple;
        multiple /= 10;
    }
}
else
{
    // 数字的起始
    values.push(c - '0');
    // 重置变量
    prev_was_digit = true;
    multiple = 10;
}
```

另外在主`if`语句中，还要加入对`.`的处理：

```
else if (c == '.')
{
    multiple = .1;
    // 特别处理以.开头的数字
    if(!prev_was_digit)
        values.push(0);
    // 这里将小数点视为特殊的digit
    prev_was_digit = true;
}
```

至此，只要将其他地方每处符号处理重置`prev_was_digit`就完成代码的修改了。

## 运行主程序并测试

```
int main()
{
    string expression;
    cout << "输入多行以'='结尾的表达式，未知的运算符将被忽略: \n";
    while(true)
    {
        getline(cin, expression);
        if(expression == string("="))
```

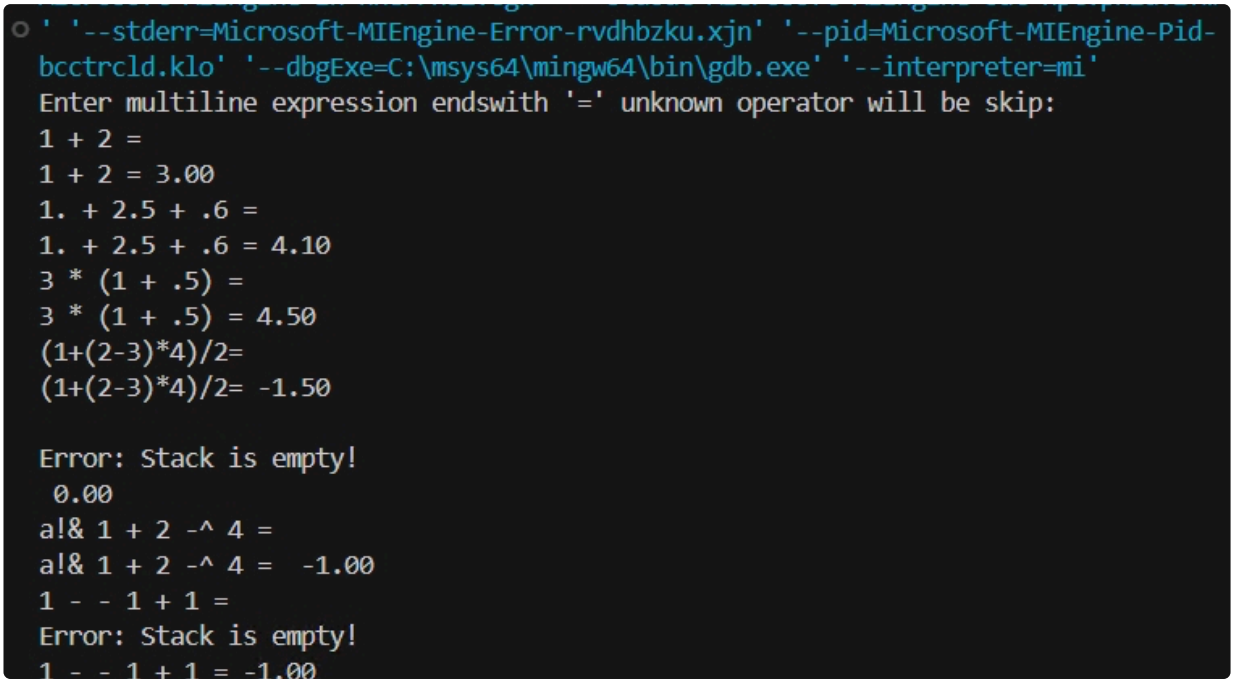
```
        {
            break;
        }

        double result = evaluate_expression(expression);
        cout << expression << " " << fixed << setprecision(2) << result << endl;
    }
    cout << "Program exited" << endl;

    return 0;
}
```

其实这个程序并没有对题目要求的结尾等号进行处理，只是都作为特殊符号忽略了。

## 测试

我们将进行一些多方面的测试：

- 测试整数的计算；
- 测试浮点数的计算；
- 测试括号内的计算；
- 测试复杂多层括号与运算符的计算；
- 不规范或异常的输入；
  - 空输入；
  - 输入中包含非法字符；
  - 输入末尾不为=；
  - 输入包含多余的运算符；
  - 输入包含多余的括号；




可以看到，我们的程序可以很好的处理正确的输入，但是对于异常的输入，程序会有一些异常的行为，有待完善。

## 结论

通过本次实验，我们成功利用栈数据结构实现了表达式求值的功能。该程序能够正确计算包含二元运算符、括号和浮点数的算术表达式。我们分别采用了栈来存储操作数和操作符，并根据操作符优先级合理地对栈进行操作，从而得到最终的计算结果。

该程序的设计思路清晰，算法实现也相对简单直观。通过对浮点数的特殊处理，我们扩展了程序的适用范围，不再局限于整数运算。测试阶段的多种异常输入情况也验证了程序的可靠性。

不过，该程序在处理某些异常输入时表现还有待改进，如空输入、多余运算符和括号等情况下的行为需要进一步优化。另外，该程序目前只支持二元运算符，对于其他运算符的处理也可以作为后续的拓展方向。

总的来说，这个实验很好地体现了栈数据结构在表达式求值等问题中的应用，对于数据结构课程的学习和编程能力的培养都有一定的帮助作用。