数据结构课程设计

项目说明文档

表达式计算

|  |  |
| --- | --- |
| 作者姓名： | 杨烜赫 |
| 学 号： | 2252709 |
| 指导教师： | 张 颖 |
| 学院专业： | 软件学院 软件工程 |



同济大学

Tongji University

1. **题目简介**

**1.实验背景**

表达式求值是程序设计语言编译中的一个最基本问题，就是将一个表达式转化为逆波兰表达式并求值。具体要求是以字符序列的形式从终端输入语法正确的，不含变量的整数表达式，并利用给定的优先关系实现对算术四则混合表达式的求值，并延时在求值过程中运算符栈，操作数栈，输入字符和主要操作变化过程。

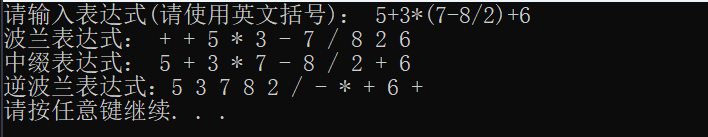
要把一个表达式翻译成正确求值的一个机器指令序列，或者直接对表达式求值，首先要能正确解释表达式。任何一个表达式都是由操作符，运算符和界限符组成，我们称它们为单词。一般来说，操作数既可以是常数，又可以是被说明为变量或常量的标识符；运算符可以分成算术运算符，关系运算符和逻辑运算符3类；基本界限符有左右括号和表达式结束符等。为了叙述的简洁，我们仅仅讨论简单算术表达式的求值问题。这种表达式只包括加，减，乘，除4种运算符。

人民在书写表达式时通常采用的是“中缀”表达形式，也就是将运算符放在两个操作数中间，用这种“中缀”形式表示的表达式称为中缀表达式。但是，这种表达式表示形式对计算机处理来说是不大合适的。对于表达式的表示还有另一种形式，称之为“后缀表达式“，也就是将运算符紧跟在两个操作书的后面。这种表达式比较合适计算机的处理方式，因此要用计算机来处理，计算表达式的问题，首先要将中缀表达式转化成后缀表达式，又称为逆波兰表达式。

**2.实验功能**

为了实现表达式求值，本项目要求首先读入表达式（包括括号）并创建对应二叉树，其次对二叉树进行前序遍历，中序遍历，后续遍历，输出对应的逆波兰式，中序表达式和波兰表达式。

**3.项目示例**

****

1. **项目设计**

**1.数据结构设计**

**enum Node\_Type**

定义了节点类型的枚举，包括 Number（数值）和 Operator（操作符）。

**struct Node\_Info**

节点信息结构体，包含节点的具体数据：

double num\_value：数值，用于数值节点。

char op\_char：操作符，用于操作符节点。

Node\_Type type：节点类型，表明节点是数值还是操作符。

**struct Expr\_Tree\_Node**

表达式树节点结构体，表示表达式树的每个节点：

Expr\_Tree\_Node\* left\_child 和 Expr\_Tree\_Node\* right\_child：指向左、右子节点的指针。

Node\_Info data：节点信息，包含数值或操作符以及节点类型。

**2.成员与操作设计**

bool is\_operator(char c)

检查给定字符是否为操作符。

int precedence(char op)

返回操作符的优先级。

void infix\_to\_postfix(char infix[], char postfix[])

将中缀表达式转换为后缀表达式。

Expr\_Tree\_Node\* construct\_tree(char postfix[])

根据后缀表达式构建表达式树。

void pre\_order(Expr\_Tree\_Node\* node)

对表达式树进行前序遍历并打印结果。

void in\_order(Expr\_Tree\_Node\* node, bool is\_root = true)

对表达式树进行中序遍历并打印结果。

void post\_order(Expr\_Tree\_Node\* node)

对表达式树进行后序遍历并打印结果。

## 3.算法设计

**1. 中缀表达式到后缀表达式的转换（infix\_to\_postfix）**

目的: 将用户输入的中缀表达式转换为后缀表达式。

实现: 使用栈来暂存操作符，根据操作符的优先级和括号来调整输出顺序。遇到操作符时，比较栈顶操作符和当前操作符的优先级，根据需要弹出栈顶操作符。

关键步骤:

如果遇到数字，直接输出到后缀表达式。

遇到左括号，将其压入栈。

遇到右括号，弹出栈中元素直到遇到左括号。

遇到操作符，根据优先级决定是否弹出栈顶元素。

**2. 构建表达式树（construct\_tree）**

目的: 根据后缀表达式构建表达式树。

实现: 使用栈来构建树。遍历后缀表达式，对于每个操作数创建树节点并入栈，对于每个操作符创建树节点，并从栈中弹出两个节点作为其子节点，再将该操作符节点入栈。

关键步骤:

遇到数字，创建数值节点并入栈。

遇到操作符，创建操作符节点，从栈中弹出两个节点作为其子节点。

**3. 表达式树的遍历（pre\_order，in\_order，post\_order）**

目的: 按不同顺序（前序、中序、后序）遍历表达式树，并打印每个节点的值。

实现: 使用递归方法实现三种遍历。

关键步骤:

前序遍历（pre\_order）: 访问根节点 -> 遍历左子树 -> 遍历右子树。

中序遍历（in\_order）: 遍历左子树 -> 访问根节点 -> 遍历右子树。在中序遍历中，还需考虑操作符的优先级来决定是否需要添加括号。

后序遍历（post\_order）: 遍历左子树 -> 遍历右子树 -> 访问根节点。

1. **项目实现**

**3.1. 中缀表达式到后缀表达式的转换**

1. **void** infix\_to\_postfix(**char** infix[], **char** postfix[]) {
2. **char** stack[MAX\_SIZE]; // 栈存储操作符
3. **int** top = -1;         // 栈顶指针
4. **int** index = 0;        // 后缀表达式的当前索引
6. **for** (**int** i = 0; infix[i] != '\0'; ++i) {
7. **char** c = infix[i];
9. **if** (isdigit(c) || c == '.') {
10. postfix[index++] = c; // 直接添加数字到后缀表达式
11. }
12. **else** **if** (c == '(') {
13. stack[++top] = c; // 左括号入栈
14. }
15. **else** **if** (c == ')') {
16. **while** (top != -1 && stack[top] != '(') {
17. postfix[index++] = ' '; // 添加空格分隔符
18. postfix[index++] = stack[top--]; // 添加栈中操作符到后缀表达式
19. }
20. top--; // 弹出 '('
21. }
22. **else** {
23. postfix[index++] = ' '; // 添加空格分隔符
24. **while** (top != -1 && precedence(stack[top]) >= precedence(c)) {
25. postfix[index++] = stack[top--]; // 弹出栈中优先级更高的操作符
26. postfix[index++] = ' ';
27. }
28. stack[++top] = c; // 当前操作符入栈
29. }
30. }
32. **while** (top != -1) { // 栈中剩余操作符添加到后缀表达式
33. postfix[index++] = ' ';
34. postfix[index++] = stack[top--];
35. }
36. postfix[index] = '\0'; // 字符串结束符
37. }

函数 infix\_to\_postfix 将给定的中缀表达式转换为后缀表达式。它通过遍历中缀表达式中的每个字符，并使用一个栈来临时存储操作符来实现这一转换。当遇到数字或小数点时，函数直接将其添加到后缀表达式中。遇到左括号时，将其推入栈中。遇到右括号时，则从栈中弹出操作符，直到遇到左括号。对于其他操作符，函数先根据优先级从栈中弹出操作符添加到后缀表达式中，然后将当前操作符推入栈中。遍历完成后，将栈中剩余的操作符依次弹出，添加到后缀表达式中。这样，中缀表达式就被转换为了后缀表达式，后者更适合进行计算机处理和求值。

**3.2. 构建表达式树功能**

1. Expr\_Tree\_Node\* construct\_tree(**char** postfix[]) {
2. Expr\_Tree\_Node\* stack[MAX\_SIZE]; // 栈存储表达式树节点
3. Expr\_Tree\_Node\* node;
4. **int** top = -1;
6. **for** (**int** i = 0; postfix[i] != '\0'; ++i) {
7. **char** c = postfix[i];
9. **if** (c == ' ') **continue**; // 忽略空格
11. **if** (!is\_operator(c)) {
12. **char** number[20]; // 存储数字字符串
13. **int** j = 0;
14. // 读取完整的数字（包括小数点）
15. **while** (isdigit(postfix[i]) || postfix[i] == '.') {
16. number[j++] = postfix[i++];
17. }
18. number[j] = '\0';
19. node = **new** Expr\_Tree\_Node(atof(number)); // 创建数字节点
20. stack[++top] = node; // 节点入栈
21. }
22. **else** {
23. node = **new** Expr\_Tree\_Node(c); // 创建操作符节点
24. node->right\_child = stack[top--]; // 设置右子节点
25. node->left\_child = stack[top--];  // 设置左子节点
26. stack[++top] = node; // 节点入栈
27. }
28. }
29. **return** stack[top]; // 返回树的根节点
30. }

函数 construct\_tree 用于从给定的后缀表达式构建一个表达式树。这个过程中，函数遍历后缀表达式中的每个字符，并使用一个栈来构建表达式树。对于每个遇到的数字，函数会创建一个表示数值的树节点并将其压入栈中。遇到操作符时，函数创建一个操作符节点，并从栈中弹出两个节点作为该操作符节点的右子节点和左子节点（注意后缀表达式的特性，先弹出的是右子节点）。这个新的操作符节点随后被压回栈中。这个过程一直持续到整个后缀表达式被遍历完毕，最终栈顶元素即为构建的表达式树的根节点。通过这种方式，函数能够将线性的后缀表达式转换为树形结构，方便进行进一步的数学操作和可视化展示。

**3.3. 表达式输出功能实现**

1. // 前序遍历打印表达式树
2. **void** pre\_order(Expr\_Tree\_Node\* node) {
3. **if** (node == nullptr) **return**;
4. **if** (node->data.type == Operator) cout << node->data.op\_char << " ";
5. **else** cout << node->data.num\_value << " ";
6. pre\_order(node->left\_child);
7. pre\_order(node->right\_child);
8. }
10. // 中序遍历打印表达式树
11. **void** in\_order(Expr\_Tree\_Node\* node, **bool** is\_root = **true**) {
12. **if** (node == nullptr) **return**;
13. **bool** needs\_parentheses = !is\_root && node->data.type == Operator &&
14. node->left\_child && node->left\_child->data.type == Operator &&
15. precedence(node->data.op\_char) > precedence(node->left\_child->data.op\_char);
17. **if** (needs\_parentheses) cout << "(";
19. in\_order(node->left\_child, **false**);
21. **if** (node->data.type == Operator) cout << node->data.op\_char << " ";
22. **else** cout << node->data.num\_value << " ";
24. in\_order(node->right\_child, **false**);
26. **if** (needs\_parentheses) cout << ")";
27. }
29. // 后序遍历打印表达式树
30. **void** post\_order(Expr\_Tree\_Node\* node) {
31. **if** (node == nullptr) **return**;
32. post\_order(node->left\_child);
33. post\_order(node->right\_child);
34. **if** (node->data.type == Operator) cout << node->data.op\_char << " ";
35. **else** cout << node->data.num\_value << " ";
36. }

这段代码实现了对表达式树的前序、中序和后序遍历，并将遍历结果打印出来。在前序遍历（pre\_order）中，程序首先访问并打印根节点的数据（操作符或数值），然后递归地对左子树和右子树进行前序遍历。在中序遍历（in\_order）中，程序先对左子树进行中序遍历，然后访问并打印根节点，最后对右子树进行中序遍历；在这里，还考虑了操作符优先级来决定是否需要在打印时添加括号以保持运算顺序正确。后序遍历（post\_order）则首先递归地对左右子树进行后序遍历，最后访问并打印根节点。这三种遍历方法分别提供了不同的视角来观察和分析表达式树的结构，对于理解和可视化表达式树至关重要。

**3.4. 主菜单功能实现**

1. **int** main() {
2. **char** infix[MAX\_SIZE], postfix[MAX\_SIZE];
4. cout << "请输入表达式(请使用英文括号)： ";
5. cin.getline(infix, MAX\_SIZE);
7. infix\_to\_postfix(infix, postfix);
9. Expr\_Tree\_Node\* root = construct\_tree(postfix);
10. cout << "波兰表达式： ";
11. pre\_order(root);
12. cout << endl << "中缀表达式： ";
13. in\_order(root);
14. cout << endl << "逆波兰表达式：";
15. post\_order(root);
16. cout << endl;
17. system("pause");
18. **return** 0;
19. }

**健全性测试：**

本程序在健全性方面表现出以下特点：

**错误处理:**

在 construct\_tree 函数中，处理了栈为空的情况，避免了对空栈执行操作导致的运行时错误。

在遍历函数（pre\_order, in\_order, post\_order）中，通过检查节点是否为 nullptr 来防止空指针引用。

**内存管理:**

程序在构建表达式树时动态分配了内存（使用 new）。虽然没有直接的内存释放操作，但这主要是因为程序在单个运行周期内构建并使用树，且随程序结束而自动释放内存。

**输入验证:**

在将中缀表达式转换为后缀表达式的过程中，程序检查每个字符以确定其类型（操作符、数字、括号），这保证了只有合法字符参与到表达式的转换中。

边界条件考虑:

在遍历函数中，考虑了节点为空的情况，这是递归遍历过程中的重要边界条件。

在 in\_order 遍历中，通过添加额外的括号来处理操作符优先级，确保了表达式的正确性。

**代码健壮性:**

通过使用递归方法来遍历表达式树，代码对于不同大小和形状的树都能正常工作，显示了良好的健壮性。

1. **项目创新点**

本项目的创新之处在于其独特的方法将数学表达式从中缀形式转换为后缀形式，并进一步构建出表达式树，这为理解和计算复杂数学表达式提供了一个直观且有效的框架。特别地，程序在中序遍历时巧妙地处理操作符优先级，通过在必要时添加括号来确保表达式的正确性。此外，通过实现前序、中序和后序三种遍历方法，程序提供了不同视角来观察和分析表达式树的结构，这对于学习和理解数据结构特别有价值。整个系统体现了算法和数据结构设计的紧密结合，展示了如何有效地处理和表示数学表达式。

**五、实验小结**

此实验成功展示了如何将中缀表达式转换为后缀表达式，并基于此构建表达式树，这不仅加深了对表达式解析的理解，也提升了对树结构和递归算法的认识。实验过程中，对错误处理的关注增强了代码的健壮性，而多种树遍历方法的实现则加深了对树遍历算法的理解。尽管程序在其功能和设计上表现出色，但对于进一步的改进，如内存管理优化和用户界面增强，还存在一定的空间。总的来说，该项目不仅实现了其主要目标，也为进一步探索数据结构和算法提供了坚实的基础。