同济大学

TONGJI UNIVERSITY

数据结构课程设计

项目名称	表达式转换
学　　院	计算机科学与技术学院
专　　业	软件工程
学生姓名	杨瑞晨
学　　号	2351050
指导教师	张颖
日　　期	2024 年 12 月 4 日

目 录

1 项目分析 …… 1
1.1 项目背景分析 …… 1
1.2 项目功能分析 …… 1
1.2.1 功能要求 …… 1
1.2.2 输入要求 …… 1
1.2.3 输出要求 …… 1
1.2.4 项目实例 …… 1
2 项目设计 …… 2
2.1 数据结构设计 …… 2
2.1.1 栈（Stack） …… 2
2.1.2 树（Tree） …… 2
2.2 类设计 …… 2
2.2.1 Stack 模板类 …… 2
2.2.2 ExpressionTree 类 …… 3
3 项目实现 …… 6
3.1 流程表示 …… 6
3.2 功能实现 …… 6
3.2.1 构建表达式树（buildTree） …… 6
3.2.2 创建子树（createSubTree） …… 9
3.2.3 辅助函数 …… 9
3.2.4 遍历方式 …… 10
3.3 main 函数 …… 11
4 项目测试 …… 12
4.1 正常测试 6 种运算符 …… 12
4.2 嵌套括号 …… 12
4.3 运算数超过 1 位整数且有非整数出现 …… 12
4.4 运算数有正负号 …… 12
4.5 只有 1 个数字 …… 13
5 项目心得与体会 …… 14

1 项目分析

1.1 项目背景分析

在计算机科学中，表达式的转换是一个常见的问题，涉及到将中缀表达式转换为前缀表达式（波兰表达式）和后缀表达式（逆波兰表达式）。这种转换不仅有助于理解不同表达式格式之间的差异，也是编译器设计中的一个关键步骤。本项目旨在通过程序模拟这一过程，实现中缀表达式到前缀和后缀表达式的转换。

1.2 项目功能分析

1.2.1 功能要求

算数表达式有前缀表示法，中缀表示法和后缀表示法等形式。日常使用的算术表达式是采用中缀表示法，即二元运算符位于两个运算数中间。请设计程序将中缀表达式转换成为后缀表达式。

1.2.2 输入要求

输入在一行中给出以空格分隔不同对象的中缀表达式，可包含 +, -, *, /, -, *, /以及左右括号，表达式不超过 20 个字符（不包括空格）

1.2.3 输出要求

在一行中输出转换后的后缀表达式，要求不同对象（运算数，运算符号）之间以空格分隔，但是结尾不得有多余空格

1.2.4 项目实例

序号	输入	输出	说明
1	2 + 3 * (7 − 4) + 8 / 4	2 3 7 4 - * + 8 4 / +	正常测试 6 种运算符
2	((2 + 3) * 4 − (8 + 2)) / 5	2 3 + 4 * 8 2 + - 5 /	嵌套括号
3	1314 + 25.5 * 12	1314 25.5 12 * +	运算数超过 1 位整数且有非整数出现
4	-2 * (+3)	-2 3 *	运算数有正或负号
5	123	123	只有 1 个数字

2 项目设计

2.1 数据结构设计

根据对题目的分析，在本项目中，我们采用了两种核心的数据结构：栈（Stack）和树（Tree）。

2.1.1 栈（Stack）

栈是一种后进先出（LIFO）的数据结构，它在本项目中用于存储操作符和临时节点，以便在构建表达式树时处理操作符的优先级，具有以下优点：

- 栈提供的基本操作如入栈（push）、出栈（pop）、查看栈顶元素（top）等操作都非常简单，易于实现。
- 栈可以在需要时动态扩容，这使得它在处理不同长度的表达式时具有很好的内存效率。
- 栈的使用大大简化了表达式转换算法的实现，尤其是在处理括号和操作符优先级时。

2.1.2 树（Tree）

表达式树是一种树形数据结构，用于表示表达式的结构。在本项目中，它用于存储和转换中缀表达式到前缀和后缀表达式。

- 清晰的表达式结构：表达式树清晰地表示了表达式的结构，包括操作符和操作数之间的关系。
- 便于遍历：表达式树便于实现前序、中序和后序遍历，这些遍历是转换表达式格式的基础。

2.2 类设计

2.2.1 Stack 模板类

Stack 模板类提供了栈的基本操作。以下是 Stack 类提供的功能：

（1）push: 将元素添加到栈顶。

（2）pop: 移除栈顶元素，并返回其值。

（3）top: 返回栈顶元素的引用。

（4）isEmpty: 检查栈是否为空。

（5）size: 返回栈中元素的数量。

```
template <typename T>
class Stack
{
private:
    T *data;      // 栈元素数组
    int topIndex; // 栈顶索引
    int capacity; // 栈容量

    void resize(int newCapacity)
    {
```

```
        // 重新分配更大的数组
        T *newData = new T[newCapacity];
        for (int i = 0; i < topIndex; ++i)
        {
            newData[i] = data[i];
        }
        delete[] data;
        data = newData;
        capacity = newCapacity;
    }

public:
    Stack(int initialCapacity = DEFAULT_CAPACITY) : topIndex(0), capacity(initialCapacity) {
    ↪  data = new T[capacity]; } // 构造函数

    ~Stack() { delete[] data; } // 析构函数

    void push(const T &value) // 入栈
    {
        // 如果栈满，扩容
        if (topIndex == capacity) resize(2 * capacity);
        data[topIndex++] = value; // 入栈
    }

    void pop() // 出栈
    {
        // 如果栈空，抛出异常
        if (isEmpty()) throw std::out_of_range("Stack underflow");
        --topIndex; // 出栈
    }

    T &top() // 获取栈顶元素
    {
        // 如果栈空，抛出异常
        if (isEmpty()) throw std::out_of_range("Stack is empty");
        return data[topIndex - 1]; // 返回栈顶元素
    }
    // 判断栈是否为空
    bool isEmpty() const { return topIndex == 0; }
    // 获取栈大小
    int size() const { return topIndex; }

    friend ostream &operator<<(ostream &out, const Stack<T> &stack)
    {
        for (int i = 0; i < stack.topIndex; ++i) out << stack.data[i] << " ";
        return out;
    }
};
```

2.2.2 ExpressionTree 类

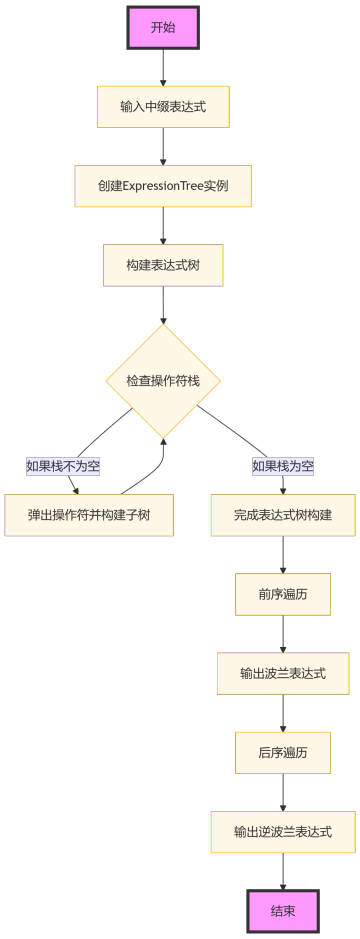
核心类，负责将输入的表达式转换为表达式树。它定义了构建树的方法 buildTree，以及遍历树的方法 preorder、inorder、postorder 来分别输出波兰表达式、中序表达式和逆波兰表达式。类中还包含一个 precedence 函数来确定操作符的优先级，这在构建正确的表达式树时至关重要。

```cpp
// 表达式树
class ExpressionTree
{
private:
    struct TreeNode
    {
        char *data;           // 数据
        TreeNode *left;       // 左孩子
        TreeNode *right;      // 右孩子
        bool hasLeftBracket;  // 是否有左括号
        bool hasRightBracket; // 是否有右括号
        TreeNode(char *data, TreeNode *left = nullptr, TreeNode *right = nullptr) :
        ↪  data(data), left(left), right(right), hasLeftBracket(false),
        ↪  hasRightBracket(false) {}
    };

    TreeNode *root; // 根节点

    // 运算符优先级
    int precedence(char op)
    {
        if (op == '+' || op == '-')
            return 1;
        else if (op == '*' || op == '/' || op == '%')
            return 2;
        else return 0;
    }

    void clear(TreeNode *node) // 清空树
    {
        if (node == nullptr)
        {
            return;
        }
        clear(node->left);
        clear(node->right);
        delete node;
    }
    // 递归构建表达式树
    TreeNode buildTree(const char *input);
    // 创建子树
    void createSubTree(Stack<TreeNode *> &nodes, Stack<char> &optrs);

    void preorder(TreeNode *node) ;  // 前序遍历
    void inorder(TreeNode *node) ;   // 中序遍历
    void postorder(TreeNode *node) ; // 后序遍历

public:
    ExpressionTree(const char *input) : root(nullptr)
    {
        root = new TreeNode(nullptr);
        *root = buildTree(input);
    }

    void printPreorder() // 前序遍历
    {
        cout << " 波兰表达式: ";
        preorder(root);
        cout << endl;
```

```
58      }
59
60      void printPostorder() // 后序遍历
61      {
62          cout << " 逆波兰表达式: ";
63          postorder(root);
64          cout << endl;
65      }
66  };
```

3 项目实现

3.1 流程表示



3.2 功能实现

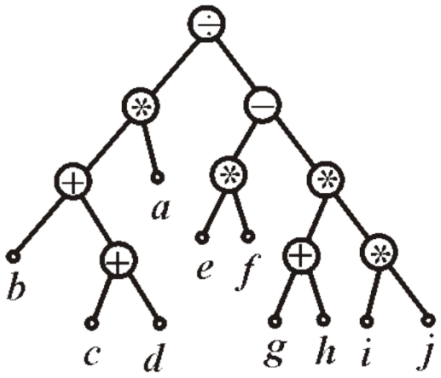
3.2.1 构建表达式树（buildTree）

ExpressionTree 类的 buildTree 方法是构建表达式树的核心函数。以下是详细的构建过程：

（1） 初始化栈：创建两个栈，nodes 用于存储 TreeNode* 类型的节点，optrs 用于存储操作符 char 类型的数据。（2） 从左到右遍历输入的中缀表达式字符串。如果当前字符是空格，则跳过如果是数字，则根据数字形式正确处理整数、负数、浮点数等。将解析出的数字转换为字符串，并创建一个新的 TreeNode 节点，将该节点推入 nodes 栈。（3） 如果当前字符是左括号 (，则将其推入 optrs 栈，并标记下一个数字节点有左括号；如果当前字符是右括号)，则继续弹出 optrs 栈顶的操作符，并与 nodes 栈顶的节点结合，形成子树，直到遇到左括号。左括号弹出后，将当前节点标记为有右括号。（4） 如果当前字符是操作符（加、减、乘、除、取模），则比较其与 optrs 栈顶操作符

的优先级。如果栈顶操作符优先级高于或等于当前操作符，或者栈为空，则将当前操作符推入 optrs 栈。如果当前操作符优先级更高，则弹出栈顶操作符，并与 nodes 栈顶的节点结合，形成子树，直到可以推入当前操作符。(5)　继续处理直到输入字符串结束，确保所有剩余的操作符都被弹出，并与节点结合形成完整的表达式树，返回根节点

以下是一个表达式树的示意图:



代码实现:

```
TreeNode buildTree(const char *input)
{
    Stack<TreeNode *> nodes;
    Stack<char> optrs;
    bool nextHasLeftBracket = false;

    int i = 0;
    int n = strlen(input);
    while (i < n)
    {
        if (input[i] == ' ')
        { // 跳过空格
            ++i;
            continue;
        }

        if (isdigit(input[i]) || ((input[i] == '-' || input[i] == '+') && isdigit(input[i + 1])))
        {
            double num = 0;

            int negtiveFlag = 1;     // 负数标志 是负数置 -1
            bool doubleFlag = false; // 小数标志
            double decimal = 1;      // 小数位数

            if (input[i] == '-')
            {
                negtiveFlag = -1;
                ++i;
            }
            else if (input[i] == '+')
            {
```

```
                ++i;
            }

            while (i < n && (isdigit(input[i]) || input[i] == '.'))
            {
                if (input[i] == '.')
                {
                    doubleFlag = true;
                    ++i;
                    continue;
                }
                if (!doubleFlag)
                {
                    num = num * 10 + input[i] - '0'; // 计算整数部分
                }
                else
                {
                    decimal *= 0.1;
                    num += (input[i] - '0') * decimal; // 计算小数部分
                }
                ++i;
            }
            // nums.push(num * negtiveFlag); // 入栈
            char *numStr = numToString(num * negtiveFlag);
            TreeNode *node = new TreeNode(numStr);
            node->hasLeftBracket = nextHasLeftBracket;
            nextHasLeftBracket = false;
            nodes.push(node); // 将数字推入栈
            ++i;
        }
        else if (input[i] == '+' || input[i] == '-' || input[i] == '*' || input[i] == '/'
        ↪  || input[i] == '%' || input[i] == '(' || input[i] == ')')
        {
            if (input[i] == '(')
            {
                nextHasLeftBracket = true;
                optrs.push(input[i]); // '(' 直接入栈
            }
            else if (input[i] == ')')
            {
                while (!optrs.isEmpty() && optrs.top() != '(') // 将 '(' 之前的运算符弹出
                {
                    createSubTree(nodes, optrs);
                }
                optrs.pop(); // 弹出 '('
                if (!nodes.isEmpty())
                {
                    nodes.top()->hasRightBracket = true;
                }
            }
            else
            {
                while (!optrs.isEmpty() && precedence(optrs.top()) >=
                ↪  precedence(input[i])) // 弹出优先级高的运算符
                {
                    createSubTree(nodes, optrs);
                }
                optrs.push(input[i]); // 入栈
            }
```

```
                ++i;
            }
        }
        // 将剩余的运算符弹出

        while (!optrs.isEmpty())
        {
            createSubTree(nodes, optrs);
        }
        return *nodes.top();
    }
```

3.2.2 创建子树（createSubTree）

用于根据操作符和节点栈中的节点创建子树。

```
    void createSubTree(Stack<TreeNode *> &nodes, Stack<char> &optrs)
    {
        char optr = optrs.top(); // 获取栈顶运算符
        optrs.pop(); // 弹出栈顶运算符
        TreeNode *right = nodes.top(); // 获取右子树
        nodes.pop(); // 弹出右子树
        TreeNode *left = nodes.top(); // 获取左子树
        nodes.pop(); // 弹出左子树
        TreeNode *curr = new TreeNode(charToString(optr)); // 创建当前运算符节点
        curr->left = left; // 设置左子树
        curr->right = right; // 设置右子树
        nodes.push(curr); // 将当前节点推入栈
    }
```

3.2.3 辅助函数

```
// 运算符优先级
int precedence(char op)
{
    if (op == '+' || op == '-')
        return 1;
    else if (op == '*' || op == '/' || op == '%')
        return 2;
    else
        return 0;
}
// 判断是否为数字
bool isdigit(char ch)
{
    return ch >= '0' && ch <= '9';
}

// 数字转字符串
char *numToString(double num)
{
    char *str = new char[STRING_SIZE];
```

```
    sprintf(str, "%g", num); // 使用 sprintf 将数字转换为字符串
    return str;
}

// 字符转字符串
char *charToString(char ch)
{
    char *str = new char[2];
    str[0] = ch;
    str[1] = '\0';
    return str;
}
```

3.2.4 遍历方式

- 前序遍历：先访问根节点，再访问左子树，最后访问右子树。
- 中序遍历：先访问左子树，再访问根节点，最后访问右子树。
- 后序遍历：先访问左子树，再访问右子树，最后访问根节点。

```
    void preorder(TreeNode *node) // 前序遍历
    {
        if (node == nullptr)
            return;
        // if (node->hasLeftBracket) cout << "(";
        cout << node->data << " ";
        preorder(node->left);
        preorder(node->right);
        // if (node->hasRightBracket) cout << ")";
    }

    void inorder(TreeNode *node) // 中序遍历
    {
        if (node == nullptr)
            return;
        if (node->hasLeftBracket)
            cout << "( ";
        inorder(node->left);
        cout << node->data << " ";
        inorder(node->right);
        if (node->hasRightBracket)
            cout << ") ";
    }

    void postorder(TreeNode *node) // 后序遍历
    {
        if (node == nullptr)
            return;
        // if (node->hasLeftBracket) cout << "(";
        postorder(node->left);
        postorder(node->right);
        cout << node->data << " ";
        // if (node->hasRightBracket) cout << ")";
    }
```

```
35
```

3.3 main 函数

```
1  int main()
2  {
3      cout << " 请输入中缀表达式（以空格分隔不同对象，不超过 20 个字符): " << endl;
4      char input[STRING_SIZE];
5      cin.getline(input, STRING_SIZE, '\n');
6
7      ExpressionTree et(input);
8      et.printPreorder();
9      et.printPostorder();
10
11     return 0;
12 }
```

4 项目测试

4.1 正常测试 6 种运算符

测试用例：2 + 3 * (7 - 4) + 8 / 4

预期结果：2 3 7 4 - * + 8 4 / +

测试结果：

```
请输入中缀表达式(以空格分隔不同对象，不超过 20 个字符):
2 + 3 * ( 7 - 4 ) + 8 / 4
波兰表达式: + + 2 * 3 - 7 4 / 8 4
逆波兰表达式: 2 3 7 4 - * + 8 4 / +
```

4.2 嵌套括号

测试用例：((2 + 3) * 4 - (8 + 2)) / 5

预期结果：2 3 + 4 * 8 2 + - 5 /

测试结果：

```
请输入中缀表达式(以空格分隔不同对象，不超过 20 个字符):
( ( 2 + 3 ) * 4 - ( 8 + 2 ) ) / 5
波兰表达式: / - * + 2 3 4 + 8 2 5
逆波兰表达式: 2 3 + 4 * 8 2 + - 5 /
```

4.3 运算数超过 1 位整数且有非整数出现

测试用例：1314 + 25.5 * 12

预期结果：1314 25.5 12 * +

测试结果：

```
请输入中缀表达式(以空格分隔不同对象，不超过 20 个字符):
1314 + 25.5 * 12
波兰表达式: + 1314 * 25.5 12
逆波兰表达式: 1314 25.5 12 * +
```

4.4 运算数有正负号

测试用例：-2 * (+3)

预期结果：-2 3 *

测试结果：

```
请输入中缀表达式(以空格分隔不同对象，不超过 20 个字符):
-2 * ( +3 )
波兰表达式: * -2 3
逆波兰表达式: -2 3 *
```

4.5 只有 1 个数字

测试用例: 65472

预期结果: 65472

测试结果:

```
请输入中缀表达式(以空格分隔不同对象，不超过 20 个字符):
65472
波兰表达式: 65472
逆波兰表达式: 65472
```

5 项目心得与体会

在完成这个表达式转换项目的编程和实现过程中，我获得了宝贵的经验和深刻的认识。

通过这个项目，我更加深刻地理解了数据结构和算法在解决实际问题中的应用。尤其是在处理表达式转换的过程中，我体会到了栈和树这两种数据结构的强大功能和灵活性。这不仅仅是对理论知识的复习，更是一次将理论应用到实践中的挑战；在编写和调试代码的过程中，我的编程技能得到了显著提升。我学会了如何设计和实现复杂的数据结构，如栈和表达式树，并且掌握了如何通过模板编程提高代码的通用性和效率。此外，我也提高了对 C++ 语言特性的掌握，如指针、动态内存管理以及异常处理；面对项目中遇到的各种问题，如括号匹配、操作符优先级处理等，我学会了如何分析问题、分解问题，并逐步找到解决方案。这个过程锻炼了我的逻辑思维能力和问题解决能力，让我在面对复杂问题时更加从容；在处理表达式的解析和转换时，我意识到细节的重要性。一个小小的语法错误或者逻辑疏漏都可能导致程序的失败。因此，我学会了在编程时更加注重细节，并且在代码中添加了更多的注释和异常处理，以提高程序的健壮性。

总之，这个项目不仅提升了我的编程技能，也锻炼了我的逻辑思维和问题解决能力。它让我更加深刻地理解了数据结构和算法的重要性，并且让我学会了如何将理论知识应用到实际问题中。这些经验和技能将为我未来的学习和工作奠定坚实的基础。