项目说明文档

数据结构课程设计

——表达式计算

作 者 姓 名　林继申

学　　号　2250758

指 导 教 师　张　颖

学 院 专 业　软件学院　软件工程

同济大学
TONGJI UNIVERSITY

二〇二三 年 十二 月 十三 日

目录

1 项目分析

1.1 项目背景分析

表达式求值是程序设计语言编译的基本问题之一，主要涉及将表达式转化为逆波兰表达式（后缀表达式）并进行求值。这一过程要求用户以字符序列的形式输入一个语法正确的整数表达式，不包含任何变量。在求值过程中，表达式的组成元素（操作符、运算符和界限符）被视作单词。这些元素包括：操作数（可能是常数或标识符）、算术运算符（加、减、乘、除）、关系运算符、逻辑运算符，以及基本界限符如括号和表达式结束符。人们通常采用中缀表达形式编写表达式，即运算符置于两个操作数之间。然而，这种表达形式并不适合计算机处理。相反，后缀表达式，即运算符紧随其操作数之后的形式，更适合计算机处理。因此，将中缀表达式转换成后缀表达式是计算机处理表达式求值问题的关键步骤。

1.2 项目需求分析

基于以上背景分析，本项目需要实现需求如下：

(1)用户应能通过命令行界面输入一个整数算术表达式。该表达式不包含变量，仅包括整数、算术运算符（加、减、乘、除）以及括号；

(2)设计简单直观的控制台界面，使操作便捷、容易上手，适应不同用户的操作习惯；

(3)读入表达式并进行解析和验证，程序需要能够解析输入的中缀表达式，并准确识别其中的数字、运算符和括号。还应有能力验证输入表达式的合法性，例如检查括号是否匹配，运算符是否正确使用等；

(4)将中缀表达式转换为后缀表达式并创建表达式二叉树，对表达式二叉树进行前序遍历，中序遍历和后序遍历，分别输出对应的前缀表达式(波兰表达式)，中缀表达式和后缀表达式（逆波兰表达式）；

(5)计算表达式的值并进行输出；

(6)实现异常处理机制，确保系统稳定性和安全性，避免因用户输入错误导致系统崩溃或信息丢失。

1.3 项目功能分析

本项目旨在通过模拟计算机处理和计算表达式的过程，实现表达式二叉树的构建，并通过前序遍历，中序遍历和后序遍历，分别输出对应的前缀表达式（波

兰表达式)，中缀表达式和后缀表达式(逆波兰表达式)，最后进行表达式的计算。下面对项目的功能进行详细分析。

1.3.1 构建表达式二叉树功能

该功能的目的是将用户输入的中缀表达式转换成二叉树形式，以便于执行后续的遍历和计算。通过二叉树的形式，可以有效地表示和处理表达式中的运算符优先级和括号结构。

1.3.2 表达式计算功能

此功能旨在通过遍历表达式二叉树来计算表达式的值。

1.3.3 异常处理功能

实现异常处理机制，处理用户可能输入的非法信息，确保系统的稳定性和安全性。

2 项目设计

2.1 数据结构设计

基于项目分析，在模拟计算机处理和计算表达式的过程中选择使用二叉树作为数据结构，主要基于以下几个考虑：

(1)表达式的层级结构：二叉树具有天然的层级结构，非常适合表示数学表达式中的操作符和操作数之间的关系。每个节点可以代表一个操作符或操作数，其子节点代表与之相关的表达式部分；

(2)便于表达式的遍历和计算：二叉树结构便于实现表达式的前序、中序和后序遍历，这对于不同类型的表达式输出和求值至关重要；

(3)灵活性和扩展性：使用二叉树结构，可以轻松添加或修改表达式的组成部分；

(4)易于理解和实现：二叉树作为一种基础的数据结构，其原理和操作相对简单明了，便于程序设计和调试；

(5)算法和数据结构的成熟：二叉树是计算机科学中一个非常成熟的数据结构，有大量的研究和实现算法可以借鉴。对于开发者来说，基于二叉树的操作和维护相对简单，有利于提高系统的开发效率和稳定性。

通过上述考虑，本项目采用了模板类 `MyBinaryTree` 来实现二叉树的基本功能，以及派生类 `ExpressionTree` 来特化为表达式的处理和计算。此外，还设

计了栈（MyStack）和队列（MyQueue）来辅助表达式的解析和遍历。

2.2 结构体与类设计

2.2.1 MyLinkNode 结构体的设计

2.2.1.1 概述

MyLinkNode 结构体是一个用于构建链表节点的模板结构体。该结构体用于表示链表中的每个节点，其中包括节点存储的数据以及指向下一个节点的指针。经典的链表一般包括两个抽象数据类型（ADT）——链表结点类（LNode）与链表类（LinkList）。本项目希望链表结点类可以直接访问链表结点，所以使用 struct 而不是 class 描述链表结点类。

2.2.1.2 结构体定义

```
template <typename Type>
struct MyLinkNode {
    Type data;
    MyLinkNode<Type>* link;
    MyLinkNode(MyLinkNode<Type>* ptr = NULL) { link = ptr; }
    MyLinkNode(const Type& item, MyLinkNode<Type>* ptr = NULL)
{ data = item; link = ptr; }
};
```

2.2.1.3 数据成员

Type data：数据域，存储节点的数据

MyLinkNode<Type>* link：指针域，指向下一个节点的指针

2.2.1.4 构造函数

MyLinkNode(MyLinkNode<Type>* ptr = NULL);

构造函数，初始化指针域。

MyLinkNode(const Type& item, MyLinkNode<Type>* ptr = NULL);

构造函数，初始化数据域和指针域。

2.2.2 MyStack 类的设计

2.2.2.1 概述

该通用模板类 MyStack 用于创建和操作堆栈数据结构。堆栈是一种基本的

数据结构，它遵循先进后出（Last-In-First-Out，LIFO）的原则，允许用户推入（push）和弹出（pop）元素，元素（即链表节点）由 MyLinkNode 结构体表示，其中包含数据和指针域。此模板类 MyStack 实现了堆栈的基本功能，并提供了一些辅助方法以方便操作。

2.2.2.2 类定义

```
template <typename Type>
class MyStack {
private:
    MyLinkNode<Type>* top;
public:
    MyStack() : top(NULL) {}
    ~MyStack() { makeEmpty(); }
    bool isEmpty(void) const { return top == NULL; }
    void makeEmpty(void);
    void push(const Type& item);
    bool pop(Type& item);
    bool getTop(Type& item);
    int getSize(void) const;
};
```

2.2.2.3 私有数据成员

MyLinkNode<Type>* top：指向堆栈顶部元素的指针

2.2.2.4 构造函数

```
MyStack() : top(NULL) {}
```

默认构造函数，创建一个空的堆栈。

2.2.2.5 析构函数

```
~MyStack() { makeEmpty(); }
```

析构函数，清空堆栈并释放内存。

2.2.2.6 公有成员函数

```
bool isEmpty(void) const { return top == NULL; }
```

判断堆栈是否为空，如果堆栈为空，它将返回 true，否则返回 false。

```
void makeEmpty(void);
```

清空堆栈，释放所有分配的内存。它通过遍历堆栈的节点，释放它们，并将

堆栈的 top 指针置为 NULL 来实现。

```
void push(const Type& item);
```

用于将新元素推入堆栈。它会在堆栈的顶部创建一个新节点，将元素存储在其中，并将新节点的 link 指向旧的堆栈顶部。

```
bool pop(Type& item);
```

从堆栈中弹出顶部元素，将其存储在传入的参数中，并释放相应的节点。如果堆栈为空，它将返回 false，否则返回 true。

```
bool getTop(Type& item);
```

用于获取堆栈的顶部元素，但不弹出它。如果堆栈为空，它将返回 false，否则返回 true。

```
int getSize(void) const;
```

计算堆栈中的元素数量，通过遍历堆栈节点并计数节点的数量来实现。

2.2.3 MyQueue 类的设计

2.2.3.1 概述

该通用模板类 MyQueue 用于存储和管理数据元素，遵循先进先出（First-in-First-Out，FIFO）的原则。该队列是基于链表数据结构实现的，链表节点由 MyLinkNode 结构体表示，其中包含数据和指针域。此模板类 MyQueue 允许在队列的前端（front）添加元素，以及从队列的前端（front）移除元素。

2.2.3.2 类定义

```
template <typename Type>
class MyQueue {
private:
    MyLinkNode<Type>* front;
    MyLinkNode<Type>* rear;
public:
    MyQueue() : front(NULL), rear(NULL) {}
    ~MyQueue() { makeEmpty(); }
    bool isEmpty(void) const { return front == NULL; }
    void makeEmpty(void);
    void enQueue(const Type& item);
    bool deQueue(Type& item);
    bool getFront(Type& item) const;
    int getSize(void) const;
```

```
};
```

2.2.3.3 私有数据成员

`MyLinkNode<Type>* front`：指针指向队列的前端，即队列中的第一个元素

`MyLinkNode<Type>* rear`：指针指向队列的后端，即队列中的最后一个元素

2.2.3.4 构造函数

```
MyQueue() : front(NULL), rear(NULL) {}
```

默认构造函数，创建一个空的队列。

2.2.3.5 析构函数

```
~MyQueue() { makeEmpty(); }
```

析构函数，清空队列并释放内存。

2.2.3.6 公有成员函数

```
bool isEmpty(void) const { return front == NULL; }
```

检查队列是否为空。

```
void makeEmpty(void);
```

清空队列，释放所有元素占用的内存。

```
void enQueue(const Type& item);
```

将元素添加到队列的末尾。

```
bool deQueue(Type& item);
```

移除队列的前端元素并将其值通过引用返回。

```
bool getFront(Type& item) const;
```

获取队列的前端元素的值。

```
int getSize(void) const;
```

获取队列中元素的数量。

2.2.4 MyBinTreeNode 结构体的设计

2.2.4.1 概述

`MyBinTreeNode` 结构体是一个模板结构体，用于构建和表示一个二叉树的节点。二叉树是一种重要的数据结构，广泛应用于各种计算机科学领域。在这个结构体中，每个节点包含一个数据元素和两个指向其子节点的指针：左子节点和右子节点。

2.2.4.2 结构体定义

```
template <typename Type>
struct MyBinTreeNode {
    Type data;
    MyBinTreeNode<Type>* leftChild;
    MyBinTreeNode<Type>* rightChild;
    MyBinTreeNode() : leftChild(NULL), rightChild(NULL) {}
    MyBinTreeNode(Type item, MyBinTreeNode<Type>* l = NULL,
MyBinTreeNode<Type>* r = NULL) : data(item), leftChild(l),
rightChild(r) {}
};
```

2.2.4.3 数据成员

Type data：数据域，存储节点的数据

MyBinTreeNode<Type>* leftChild：指针域，指向左子节点的指针

MyBinTreeNode<Type>* rightChild：指针域，指向右子节点的指针

2.2.4.4 构造函数

```
MyBinTreeNode() : leftChild(NULL), rightChild(NULL) {}
```

构造函数，初始化指针域。

```
MyBinTreeNode(Type item, MyBinTreeNode<Type>* l = NULL,
MyBinTreeNode<Type>* r = NULL) : data(item), leftChild(l),
rightChild(r) {}
```

构造函数，初始化数据域和指针域。

2.2.5 MyBinaryTree 类的设计

2.2.5.1 概述

MyBinaryTree 类是一个模板类，用于创建和管理二叉树结构。二叉树是一种基础且重要的数据结构，广泛应用于多种场景，如管理、排序和搜索算法等。本模板类提供了二叉树的基本操作，包括插入、查找、遍历、修改和删除节点等功能。

2.2.5.2 类定义

```
template <typename Type>
class MyBinaryTree {
protected:
```

```
        MyBinTreeNode<Type>* root;
    private:
        typedef              void              (MyBinaryTree::*
VisitFunction)(MyBinTreeNode<Type>*);
        MyBinTreeNode<Type>*   copy(const   MyBinTreeNode<Type>*
subTree);
        void outputNode(MyBinTreeNode<Type>* node) { if (node !=
NULL) std::cout << node->data; }
    public:
        MyBinaryTree() : root(NULL) {}
        MyBinaryTree(Type& item);
        MyBinaryTree(MyBinaryTree<Type>&   other)   {   root   =
copy(other.root); }
        ~MyBinaryTree() { destroy(root); }
        void destroy(MyBinTreeNode<Type>*& subTree);
        bool isEmpty(void) { return root == NULL; }
        int  getHeight(MyBinTreeNode<Type>*  subTree)  {  return
(subTree == NULL) ? 0 : (std::max(getHeight(subTree->leftChild),
getHeight(subTree->rightChild)) + 1); }
        int  getSize(MyBinTreeNode<Type>*  subTree)  {  return
(subTree  ==  NULL)  ?  0  :  (getSize(subTree->leftChild)  +
getSize(subTree->rightChild) + 1); }
        MyBinTreeNode<Type>* getRoot(void) { return root; }
        MyBinTreeNode<Type>*     getParent(MyBinTreeNode<Type>*
current, MyBinTreeNode<Type>* subTree);
        MyBinTreeNode<Type>*  getLeftChild(MyBinTreeNode<Type>*
current) { return current == NULL ? NULL : current->leftChild; }
        MyBinTreeNode<Type>*  getRightChild(MyBinTreeNode<Type>*
current) { return current == NULL ? NULL : current->rightChild; }
        MyBinTreeNode<Type>*    findNode(const    Type&    item,
MyBinTreeNode<Type>* subTree);
        void preOrder(VisitFunction visit, MyBinTreeNode<Type>*
subTree);
        void  inOrder(VisitFunction  visit,  MyBinTreeNode<Type>*
subTree);
```

```
        void postOrder(VisitFunction visit, MyBinTreeNode<Type>*
subTree);
        void           levelOrder(VisitFunction           visit,
MyBinTreeNode<Type>* subTree);
        void    preOrderOutput(MyBinTreeNode<Type>*    subTree)
{ preOrder(&MyBinaryTree<Type>::outputNode, subTree); }
        void    inOrderOutput(MyBinTreeNode<Type>*    subTree)
{ inOrder(&MyBinaryTree<Type>::outputNode, subTree); }
        void   postOrderOutput(MyBinTreeNode<Type>*   subTree)
{ postOrder(&MyBinaryTree<Type>::outputNode, subTree); }
        void   levelOrderOutput(MyBinTreeNode<Type>*   subTree)
{ levelOrder(&MyBinaryTree<Type>::outputNode, subTree); }
        bool leftInsertNode(MyBinTreeNode<Type>* current, const
Type& item);
        bool rightInsertNode(MyBinTreeNode<Type>* current, const
Type& item);
        bool modifyNode(const Type& oldItem, const Type& newItem,
MyBinTreeNode<Type>* subTree);
        MyBinaryTree<Type>& operator=(const MyBinaryTree<Type>&
other);
    };
```

2.2.5.3 受保护的数据成员

MyBinTreeNode<Type>* root：指向树的根节点的指针

2.2.5.4 VisitFunction 类型定义

```
typedef void (MyBinaryTree::* VisitFunction)(MyBinTreeNode<Type>*);
```

在 MyBinaryTree 类中，VisitFunction 是一个特殊的类型定义，属于类定义的一部分。这是一个函数指针类型，用于指向 MyBinaryTree 类的成员函数，这些成员函数接受一个指向 MyBinTreeNode<Type>类型的指针作为参数并返回 void。

VisitFunction 类型主要用于树的遍历操作中，允许将成员函数作为参数传递给遍历函数，从而实现对树中每个节点执行特定操作的功能。通过这种方式，可以在遍历树时灵活地应用不同的处理逻辑，例如打印节点数据、计算节点总数、修改节点数据等。

2.2.5.5 构造函数

```
MyBinaryTree() : root(NULL) {}
```

默认构造函数，初始化一个空的二叉树，即根节点设置为 NULL。

```
MyBinaryTree(Type& item);
```

转换构造函数，创建一个二叉树，并设置根节点的数据为提供的 item，适用于已知根节点数据的情况，便于直接构建含有根节点的二叉树。

```
MyBinaryTree(MyBinaryTree<Type>& other) { root = copy(other.root); }
```

复制构造函数，创建一个新的二叉树，其结构和数据是另一个二叉树（other）的深拷贝。

2.2.5.6 析构函数

```
~MyBinaryTree() { destroy(root); }
```

析构函数，用于在 MyBinaryTree 类的对象不再需要时，安全地销毁该对象。它负责释放二叉树中所有节点占用的内存资源，防止内存泄漏。

2.2.5.7 私有成员函数

```
MyBinTreeNode<Type>* copy(const MyBinTreeNode<Type>* subTree);
```

深拷贝一个子树。

```
void outputNode(MyBinTreeNode<Type>* node) { if (node != NULL) std::cout << node->data; }
```

输出一个节点的数据。

2.2.5.8 公有成员函数

```
void destroy(MyBinTreeNode<Type>*& subTree);
```

递归地销毁整个子树。

```
bool isEmpty(void) { return root == NULL; }
```

检查树是否为空。

```
int getHeight(MyBinTreeNode<Type>* subTree) { return (subTree == NULL) ? 0 : (std::max(getHeight(subTree->leftChild), getHeight(subTree->rightChild)) + 1); }
```

计算树或子树的高度。

```
int getSize(MyBinTreeNode<Type>* subTree) { return (subTree == NULL) ? 0 : (getSize(subTree->leftChild) + getSize(subTree->rightChild) + 1); }
```

计算树或子树的节点总数。

```
MyBinTreeNode<Type>* getRoot(void) { return root; }
```

获取树的根节点。

```
MyBinTreeNode<Type>* getParent(MyBinTreeNode<Type>* current, MyBinTreeNode<Type>* subTree);
```

查找指定节点的父节点。

```
MyBinTreeNode<Type>* getLeftChild(MyBinTreeNode<Type>* current) { return current == NULL ? NULL : current->leftChild; }
```

获取指定节点的左子节点。

```
MyBinTreeNode<Type>* getRightChild(MyBinTreeNode<Type>* current) { return current == NULL ? NULL : current->rightChild; }
```

获取指定节点的右子节点。

```
MyBinTreeNode<Type>* findNode(const Type& item, MyBinTreeNode<Type>* subTree);
```

在树中查找包含特定数据的节点。

```
void preOrder(VisitFunction visit, MyBinTreeNode<Type>* subTree);
```

前序遍历（Pre-Order Traversal）树。先访问根节点，然后递归地进行左子树的前序遍历，最后递归地进行右子树的前序遍历。

```
void inOrder(VisitFunction visit, MyBinTreeNode<Type>* subTree);
```

中序遍历（In-Order Traversal）树。首先递归地进行左子树的中序遍历，然后访问根节点，最后递归地进行右子树的中序遍历。

```
void postOrder(VisitFunction visit, MyBinTreeNode<Type>* subTree);
```

后序遍历（Post-Order Traversal）树。首先递归地进行左子树的后序遍历，然后递归地进行右子树的后序遍历，最后访问根节点。

```
void levelOrder(VisitFunction visit, MyBinTreeNode<Type>* subTree);
```

层序遍历（Level-Order Traversal）树。按照树的层次从上到下，从左到右逐层访问每个节点。通常使用队列来实现。将根节点入队，然后在循环中不断出队一个节点并将其子节点入队，直到队列为空。

```
void preOrderOutput(MyBinTreeNode<Type>* subTree) { preOrder(&MyBinaryTree<Type>::outputNode, subTree); }
```

前序遍历树输出。

void inOrderOutput(MyBinTreeNode<Type>* subTree) { inOrder(&MyBinaryTree<Type>::outputNode, subTree); }

中序遍历树输出。

void postOrderOutput(MyBinTreeNode<Type>* subTree) { postOrder(&MyBinaryTree<Type>::outputNode, subTree); }

后续遍历树输出。

void levelOrderOutput(MyBinTreeNode<Type>* subTree) { levelOrder(&MyBinaryTree<Type>::outputNode, subTree); }

层数遍历树输出。

bool leftInsertNode(MyBinTreeNode<Type>* current, const Type& item);

在指定节点插入左子节点。

bool rightInsertNode(MyBinTreeNode<Type>* current, const Type& item);

在指定节点插入右子节点。

bool modifyNode(const Type& oldItem, const Type& newItem, MyBinTreeNode<Type>* subTree);

修改树中特定数据的节点。

2.2.5.9 运算符重载

MyBinaryTree<Type>& operator=(const MyBinaryTree<Type>& other);

重载赋值运算符，用于实现树的深拷贝。

2.2.6 ExpressionTree 类的设计

2.2.6.1 概述

ExpressionTree 类是一个专门为算术表达式处理和计算设计的类，它继承自 MyBinaryTree<char>类。该类的主要功能是将一个算术表达式转换成一个二叉树结构，并提供计算该表达式的结果的方法。它使用了两个栈（MyStack<char>和 MyStack<MyBinTreeNode<char>*>），以便于实现表达式的解析和构建二叉树。

2.2.6.2 类定义

```
class ExpressionTree : public MyBinaryTree<char> {
private:
    MyStack<char> charStack;
```

```
    MyStack<MyBinTreeNode<char>*> treeStack;
    int precedence(char op);
    double calculateRecursive(MyBinTreeNode<char>* node);
public:
    void createExpressionTree(const char expression[]);
    double calculate(void) { return calculateRecursive(this->root); }
};
```

2.2.6.3 私有数据成员

`MyStack<char> charStack`：用于存储操作符的栈

`MyStack<MyBinTreeNode<char>*> treeStack`：用于构建表达式树的节点栈

2.2.6.4 私有成员函数

```
int precedence(char op);
```

这是一个私有方法，用于确定不同操作符的优先级，支持表达式中操作符的正确解析和树的正确构建。

```
double calculateRecursive(MyBinTreeNode<char>* node);
```

这是一个私有方法，用于递归地计算表达式树中每个节点的值。

2.2.6.5 公有成员函数

```
void createExpressionTree(const char expression[]);
```

此方法接受一个字符数组作为输入，该数组代表一个数学表达式。它将表达式转换为一个二叉树结构，每个节点代表一个操作数或操作符。这种结构方便对表达式进行遍历和计算。

```
double calculate(void) { return calculateRecursive(this->root); }
```

此方法不接受任何参数，直接对已构建的表达式树进行计算，返回表达式的数值结果。它使用递归的方式遍历树，并计算每个节点代表的子表达式的值。

2.3 项目主体架构设计

项目主体架构设计为：

(1)初始化和提示信息；

(2)输入表达式并构建表达式二叉树；

(3)输出不同形式的表达式；

(4)计算表达式的值；

(5)等待用户退出。

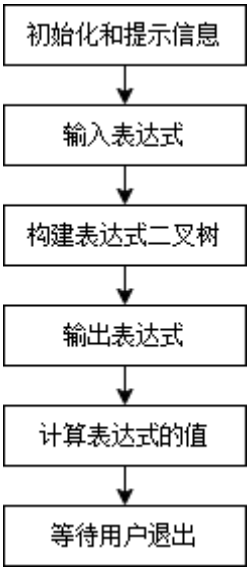

图 2.3.1 项目主体架构设计流程图

3 项目功能实现

3.1 项目主体架构的实现

3.1.1 项目主体架构实现思路

项目主体架构实现思路为：

(1)初始化和提示信息：程序开始时，首先通过 printPrompt 函数展示用户界面和输入要求，包括表达式的字符限制、有效字符种类、括号匹配要求等。这一步骤为用户提供了清晰的指导，确保输入的表达式符合程序处理的格式；

(2)输入表达式：用户被提示输入一个表达式。这个表达式被存储在一个字符数组 expression 中，其长度限制为 MAX_LENGTH。循环检查输入的表达式是否有效（通过 isValidExpression 函数）。这包括检查表达式的格式是否正确、是否存在非法字符等；

(3)构建表达式二叉树：一旦输入有效，程序使用 ExpressionTree 类的实例来创建一个表达式树。这是通过调用 createExpressionTree 方法实现的，该方法将输入的表达式转换为二叉树的形式。在这个阶段，表达式中的每个数字和操作符都被作为树的节点处理，构建出反映表达式结构的二叉树；

(4)输出不同形式的表达式：程序输出原始的中缀表达式（用户输入的形式）、转换后的前缀（波兰）表达式和后缀（逆波兰）表达式。这些输出是通过对表达式树进行前序、中序和后序遍历得到的，展示了表达式在不同格式下的表示；

(5)计算表达式的值：程序通过调用 ExpressionTree 类的 calculate 方

法计算表达式的数值结果。该方法内部使用递归遍历表达式树，根据树节点的操作符和操作数进行计算；

(6)等待用户退出：最后，程序等待用户按下回车键退出。这一步骤提供了用户与程序交互的最终环节，确保用户能够在查看结果后自主结束程序。

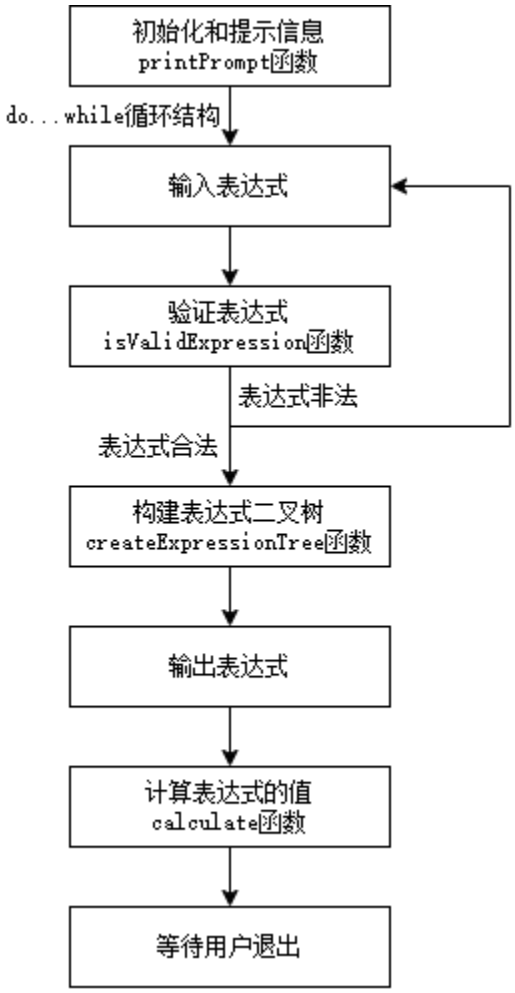

图 3.1.1.1 项目主体架构实现流程图

3.1.2 项目主体架构核心代码

```
int main()
{
    /* Enter the expression */
    printPrompt();
    char expression[MAX_LENGTH + 1] = { 0 };
    do {
        std::cout << std::endl << "请输入表达式: ";
        std::cin.getline(expression, MAX_LENGTH + 1);
        std::cout << std::endl;
    } while (!isValidExpression(expression));

    /* Build the expression tree */
```

```
        ExpressionTree expressionTree;
        expressionTree.createExpressionTree(expression);

        /* Output three types of the expression */
        std::cout << ">>> 前缀表达式（波兰表达式） : ";
        expressionTree.preOrderOutput(expressionTree.getRoot());
        std::cout << std::endl << std::endl << ">>> 中缀表达式          : " << expression << std::endl;
        std::cout << std::endl << ">>> 后缀表达式（逆波兰表达式）: ";
        expressionTree.postOrderOutput(expressionTree.getRoot());
        std::cout << std::endl << std::endl;

        /* Calculate the expression */
        std::cout << ">>> 表达式的值: " << expressionTree.calculate() << std::endl << std::endl;

        /* Wait for enter to quit */
        std::cout << "Press Enter to Quit" << std::endl;
        while (_getch() != '\r')
            continue;

        /* Program ends */
        return 0;
    }
```

3.1.3 项目主体架构示例

```
+--------------------------+
|        表达式计算        |
|  Expression Calculation  |
+--------------------------+

>>> 表达式输入要求
    [1] 表达式为不超过 256 个字符组成的字符串，超出部分将被截断
    [2] 表达式中只存在以下 16 种字符: 0 1 2 3 4 5 6 7 8 9 + - * / ( )
    [3] 表达式中的括号嵌套匹配
    [4] 表达式仅适用于单位数运算，不适用于多位数运算
    [5] 表达式中每个运算符前后必须连接数字（"-n"请输入为"0-n"）

请输入表达式: 1+2

>>> 前缀表达式（波兰表达式）  : +12

>>> 中缀表达式              : 1+2

>>> 后缀表达式（逆波兰表达式）: 12+

>>> 表达式的值: 3
```

图 3.1.3.1 项目主体架构示例

3.2 构建表达式二叉树功能的实现

3.2.1 构建表达式二叉树功能实现思路

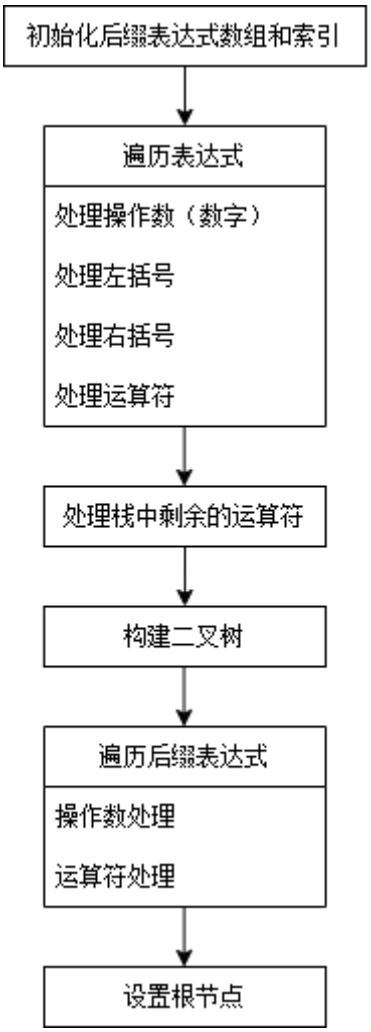

图 3.2.1.1 构建表达式二叉树功能实现流程图

构建表达式二叉树功能的函数为 ExpressionTree 类的成员函数 createExpressionTree，构建表达式二叉树功能实现的思路为：

(1)初始化后缀表达式数组和索引：创建一个字符数组来存储后缀表达式，并设置一个索引变量用于跟踪数组的当前位置；

(2)遍历表达式：对输入的中缀表达式中的每个字符进行遍历；

①处理操作数（数字）：如果当前字符是操作数（即数字），则直接将其添加到后缀表达式数组中；

②处理左括号：遇到左括号“(”时，将其推入栈中；

③处理右括号：遇到右括号“)”时，从栈中弹出元素并添加到后缀表达式数组中，直到遇到左括号“(”。然后弹出左括号；

④处理运算符：对于运算符，当栈顶运算符的优先级大于或等于当前运算符时，弹出栈顶运算符并添加到后缀表达式数组中，然后将当前运算符推入栈中；

(3)处理栈中剩余的运算符：遍历完成后，将栈中剩余的所有运算符弹出并添加到后缀表达式数组中；

(4)构建二叉树

①遍历后缀表达式：对后缀表达式数组进行遍历；

②操作数处理:遇到操作数时,为其创建一个树节点,并将该节点推入栈中；

③运算符处理:遇到运算符时,从栈中弹出两个节点(右操作数和左操作数),为这个运算符创建一个新的树节点,并将这两个弹出的节点设置为新节点的左右子节点。然后，将这个新节点推回栈中；

④设置根节点：遍历完成后，栈顶元素即为表达式树的根节点。

3.2.2 构建表达式二叉树功能核心代码

```
int ExpressionTree::precedence(char op)
{
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return 0;
}

void ExpressionTree::createExpressionTree(const char expression[])
{
    char postfix[MAX_LENGTH + 1] = { 0 }; // Array to store postfix
expression
    int postfixIndex = 0; // Index for the postfix array
    for (int i = 0; expression[i] != '\0'; i++) {
        char token = expression[i];
        if (token >= '0' && token <= '9')
            postfix[postfixIndex++] = token; // Directly add operand
to postfix
        else if (token == '(')
            charStack.push(token); // Push '(' onto stack
        else if (token == ')') {
            char topToken;
            charStack.getTop(topToken);
            while (!charStack.isEmpty() && topToken != '(') {
                charStack.pop(postfix[postfixIndex++]);
                if (!charStack.isEmpty())
                    charStack.getTop(topToken);
            }
            charStack.pop(topToken); // Pop the '(' from the stack
        }
```

```
            else {
                char topToken;
                charStack.getTop(topToken);
                while (!charStack.isEmpty() && precedence(token) <=
precedence(topToken))
                    charStack.pop(postfix[postfixIndex++]);
                charStack.push(token); // Push current operator onto stack
            }
        }
        while (!charStack.isEmpty())
            charStack.pop(postfix[postfixIndex++]); // Pop remaining
operators from the stack
        for (int i = 0; postfix[i] != '\0'; i++) {
            char token = postfix[i];
            MyBinTreeNode<char>* node;
            if (token >= '0' && token <= '9') {
                node = new(std::nothrow) MyBinTreeNode<char>(token);
                if (node == NULL) {
                    std::cerr << "Error: Memory allocation failed." <<
std::endl;
                    exit(MEMORY_ALLOCATION_ERROR);
                }
                treeStack.push(node); // Push node onto tree stack
            }
            else {
                MyBinTreeNode<char>* right, * left;
                treeStack.pop(right);
                treeStack.pop(left);
                node = new(std::nothrow) MyBinTreeNode<char>(token, left,
right);
                if (node == NULL) {
                    std::cerr << "Error: Memory allocation failed." <<
std::endl;
                    exit(MEMORY_ALLOCATION_ERROR);
                }
                treeStack.push(node); // Push the new node onto the tree
stack
            }
        }
        treeStack.getTop(this->root); // Set the root of the tree
    }
```

3.3 表达式计算功能的实现

3.3.1 表达式计算功能实现思路

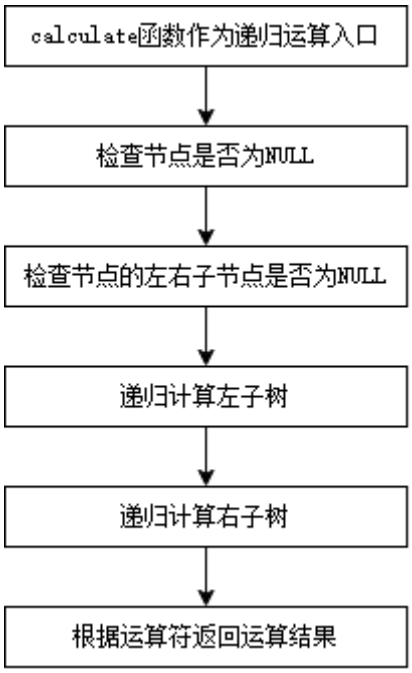

图 3.3.1.1 表达式计算功能实现流程图

表达式计算功能的函数为 ExpressionTree 类的成员函数 calculate，表达式计算功能实现的思路为：

(1)计算入口：calculate 函数作为入口，调用 calculateRecursive 函数并传入根节点，开始递归过程；

(2)递归基准情况：如果当前节点为 NULL，返回 0。如果当前节点是叶子节点（即操作数），返回其数值（字符转换为数值）；

(3)递归处理：对左子节点和右子节点递归调用 calculateRecursive 函数，得到左右子树的计算结果。根据当前节点的运算符（+，-，*，/），对左右子树的计算结果执行相应的运算；

(4)错误处理：如果遇到无效运算符，抛出无效运算符错误。如果执行除法时右操作数为零，抛出除以零错误。

3.3.2 表达式计算功能核心代码

```
double calculate(void) { return calculateRecursive(this->root); }

double ExpressionTree::calculateRecursive(MyBinTreeNode<char>* node)
{
    if (node == NULL)
        return 0;
    if (node->leftChild == NULL && node->rightChild == NULL)
```

```
            return node->data - '0';
        double leftVal = calculateRecursive(node->leftChild), rightVal =
calculateRecursive(node->rightChild);
        switch (node->data) {
            case '+':
                return leftVal + rightVal;
            case '-':
                return leftVal - rightVal;
            case '*':
                return leftVal * rightVal;
            case '/':
                if (rightVal >= -DEVIATION && rightVal <= DEVIATION) {
                    std::cerr << "Error: Division by zero." << std::endl;
                    exit(DIVISION_BY_ZERO_ERROR);
                }
                return leftVal / rightVal;
            default:
                return 0;
        }
    }
```

3.4 异常处理功能的实现

3.4.1 动态内存申请失败的异常处理

在进行 MyLinkNode 类和 MyBinTreeNode 类等的动态内存申请时，程序使用 new(std::nothrow)来尝试分配内存。new(std::nothrow)在分配内存失败时不会引发异常，而是返回一个空指针（NULL 或 nullptr），代码检查指针是否为空指针，如果为空指针，意味着内存分配失败，这时程序将执行以下操作：

(1) 向标准错误流 std::cerr 输出一条错误消息"Error: Memory allocation failed."，指出内存分配失败；

(2)调用 exit 函数，返回错误码 MEMORY_ALLOCATION_ERROR（通过宏定义方式定义为-1），用于指示内存分配错误，并导致程序退出。

下面是动态内存申请的异常处理的一个代码示例：

```
template <typename Type>
MyBinaryTree<Type>::MyBinaryTree(Type& item)
{
    root = new(std::nothrow) MyBinTreeNode<Type>(item);
    if (root == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
```

```
}
```

3.4.2 MyStack 类栈空的异常处理

在调用MyStack类的pop和getTop等成员函数时，如果栈为空（isEmpty()返回true），则函数返回false，表示栈不包含任何元素。

3.4.3 MyQueue 类队列空的异常处理

在调用MyQueue类的deQueue和getFront等成员函数时，如果队列为空（isEmpty()返回true），则函数返回false，表示队列不包含任何元素。

3.4.4 MyBinaryTree 类自赋值的异常处理

MyBinaryTree类重载了赋值运算符，用于将一个二叉树赋值给另一个二叉树。自赋值是一种常见的错误操作，可能导致资源泄漏和其他问题，该运算符重载函数对自赋值（将二叉树赋值给自身）的情况进行了处理。

程序会检查this指针和传入的other二叉树对象是否相同，如果this指针与other相同，表示自赋值操作，为了防止自赋值，运算符重载函数不会执行赋值操作，而是直接返回当前对象的引用*this。这个错误处理机制确保了当尝试将二叉树赋值给自身时，不会导致资源泄漏或其他问题。

3.4.5 ExpressionTree 类除以零错误的异常处理

在数学和计算机程序中，除以零是一个未定义的操作。尝试执行此操作通常会导致程序出错或异常情况。在ExpressionTree类的成员函数calculateRecursive中，当遇到除法运算符（/）时，会检查右操作数（除数）是否接近零。这是通过比较rightVal（右操作数的值）的绝对值是否小于等于DEVIATION来实现的，其中DEVIATION被定义为1e-6，这是一个非常小的值，用于处理浮点数计算中的精度问题。如果rightVal在这个误差范围内，程序会认为这是一个除以零的尝试，这时程序将执行以下操作：

⑴向标准错误流std::cerr输出一条错误消息"Error: Division by zero."，指出除以零错误；

⑵调用exit函数，返回错误码DIVISION_BY_ZERO_ERROR（通过宏定义方式定义为-2），用于指示除以零错误，并导致程序退出。

3.4.6 表达式输入非法的异常处理

表达式输入非法的异常处理通过如下代码实现：

```
bool isValidExpression(const char expression[])
{
    if (expression[0] == '\0') {
```

```
            std::cout << ">>> 表达式为空，请重新输入！" << std::endl;
            return false;
        }
        MyStack<char> parenthesesStack;
        bool lastWasOperator = false, lastWasOperand = false;
        for (int i = 0; expression[i] != '\0'; i++) {
            char ch = expression[i];
            if (!((ch >= '0' && ch <= '9') || ch == '+' || ch == '-' || ch
== '*' || ch == '/' || ch == '(' || ch == ')')) {
                std::cout << ">>> 表达式存在非法字符输入，请重新输入！" <<
std::endl;
                return false;
            }
            if (i == 0 && (ch == '+' || ch == '-' || ch == '*' || ch ==
'/')) {
                std::cout << ">>> 表达式不能以运算符开始，请重新输入！" <<
std::endl;
                return false;
            }
            if (ch == '(') {
                if (expression[i + 1] == ')') {
                    std::cout << ">>> 表达式存在空括号，请重新输入！" <<
std::endl;
                    return false;
                }
                parenthesesStack.push(ch);
                lastWasOperator = false;
                lastWasOperand = false;
            }
            else if (ch == ')') {
                char tmp;
                parenthesesStack.getTop(tmp);
                if (parenthesesStack.isEmpty() || tmp != '(') {
                    std::cout << ">>> 表达式括号不匹配，请重新输入！" <<
std::endl;
                    return false;
                }
                parenthesesStack.pop(tmp);
            }
            if (ch >= '0' && ch <= '9') {
                if (lastWasOperand) {
                    std::cout << ">>> 表达式仅适用于单位数运算，不适用于多位数
运算，请重新输入！" << std::endl;
                    return false;
```

```
                }
                lastWasOperand = true;
                lastWasOperator = false;
            }
            else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
                if (!lastWasOperand) {
                    std::cout << ">>> 表达式中每个运算符前后必须连接数字，请重新输入！" << std::endl;
                    return false;
                }
                lastWasOperand = false;
                lastWasOperator = true;
            }
        }
        if (!parenthesesStack.isEmpty()) {
            std::cout << ">>> 表达式括号不匹配，请重新输入！" << std::endl;
            return false;
        }
        if (lastWasOperator) {
            std::cout << ">>> 表达式不能以运算符结尾，请重新输入！" << std::endl;
            return false;
        }
        return true;
    }
```

表达式输入要求为：

(1)表达式为不超过 256 个字符组成的字符串，超出部分将被截断；

(2)表达式中只存在以下 16 种字符：0 1 2 3 4 5 6 7 8 9 + - * / ( )；

(3)表达式中的括号嵌套匹配；

(4)表达式仅适用于单位数运算，不适用于多位数运算；

(5)表达式中每个运算符前后必须连接数字（“-n”请输入为“0-n”）。

这段代码会等待用户输入表达式，并对表达式合法性进行检查。如果表达式合法，则函数返回 `true`，如果表达式不合法，则函数返回 `false`。这段代码的具体执行逻辑如下：

(1)检查空表达式：如果表达式为空（即第一个字符为终止符`'\0'`），输出提示信息并返回 `false`；

(2)字符合法性检查：遍历表达式中的每个字符。如果字符不是数字（0-9）、运算符（+，-，*，/）或括号（(，)），输出提示信息并返回 `false`；

(3)检查表达式开头：如果表达式以运算符（+，-，*，/）开头，输出提示信息并返回 `false`；

(4)处理括号和运算符逻辑：使用栈（parenthesesStack）来确保表达式中的括号正确匹配，使用标志变量 lastWasOperator 和 lastWasOperand 来追踪上一个字符是否为运算符或操作数。对于每个左括号“(”，检查后面是否直接跟随一个右括号“)”（空括号），如果是，则输出提示信息并返回 false。对于每个右括号“)”，检查栈是否为空或栈顶是否不是左括号“(”，如果是，则输出提示信息并返回 false。对于每个运算符，检查前一个字符是否不是操作数，如果是，则输出提示信息并返回 false；

(5)检查操作数的合法性：如果连续出现两个操作数（违反单位数运算的规则），输出提示信息并返回 false；

(6)结束时的检查：遍历结束后，检查是否所有的括号都已匹配，并检查表达式是否以运算符结束；

(7)返回结果：如果以上所有检查都通过，则函数返回 true，表示表达式合法；否则，在任一步骤中遇到非法情况时，函数返回 false。

4 项目测试

4.1 表达式输入合法性验证功能测试

可以验证程序对表达式输入非法的情况进行了处理。表达式输入要求为：

(1)表达式为不超过 256 个字符组成的字符串，超出部分将被截断；

(2)表达式中只存在以下 16 种字符：0 1 2 3 4 5 6 7 8 9 + - * / ( )；

(3)表达式中的括号嵌套匹配；

(4)表达式仅适用于单位数运算，不适用于多位数运算；

(5)表达式中每个运算符前后必须连接数字（“-n”请输入为“0-n”）。

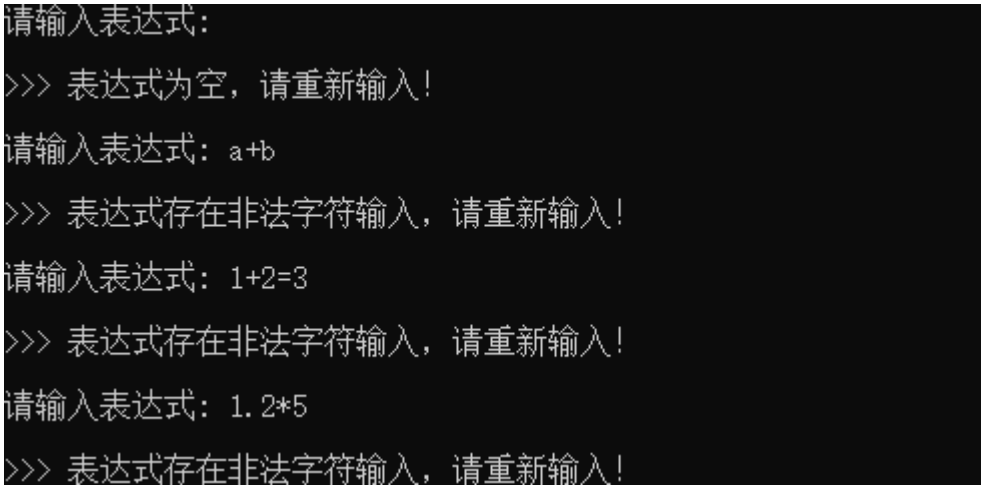

图 4.1.1 表达式输入合法性验证功能测试（测试一）

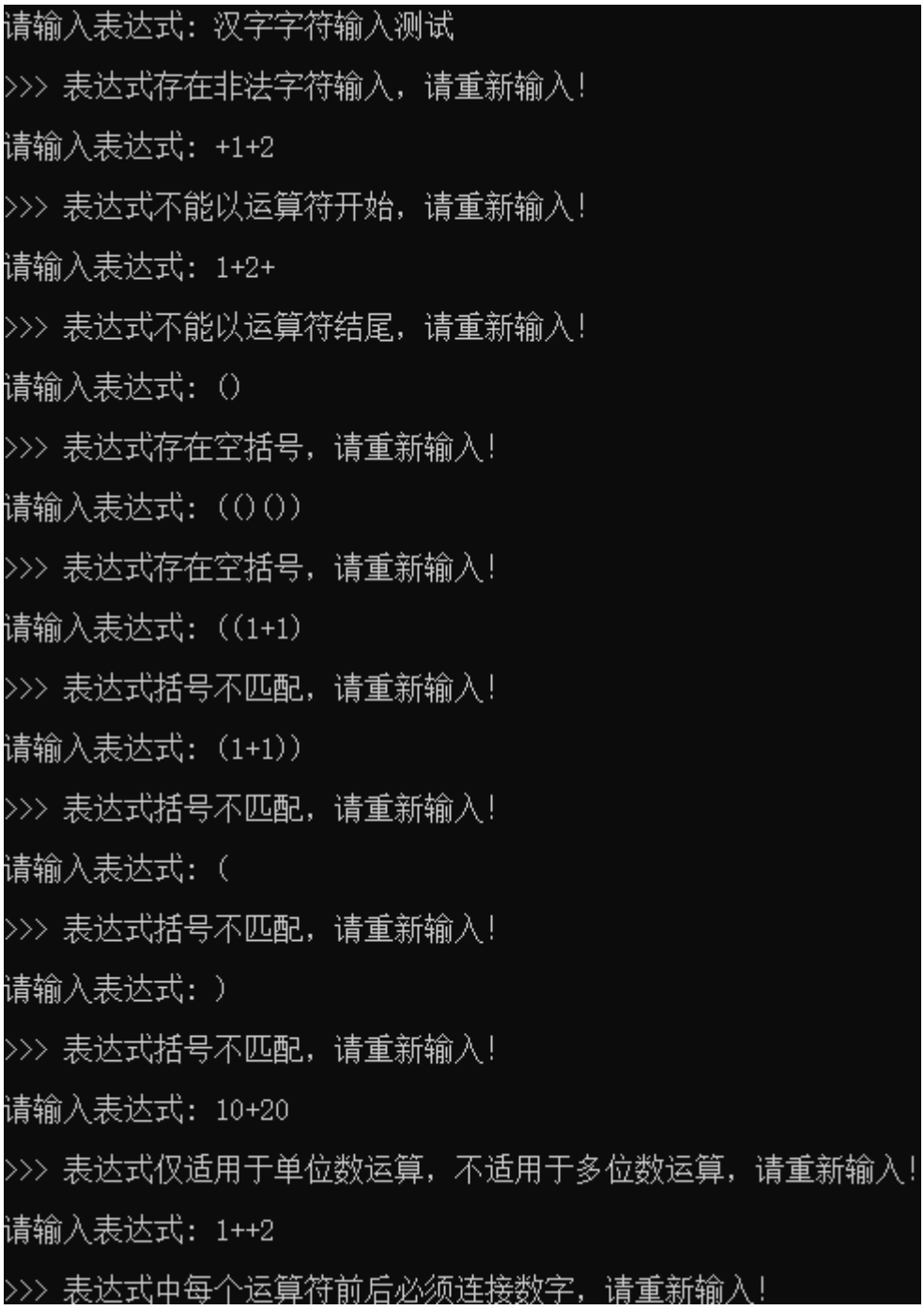

图 4.1.2 表达式输入合法性验证功能测试（测试二）

4.2 构建表达式二叉树功能和表达式计算功能测试

将中缀表达式转换为后缀表达式并创建表达式二叉树，对表达式二叉树进行前序遍历，中序遍历和后序遍历，分别输出对应的前缀表达式（波兰表达式），中缀表达式和后缀表达式（逆波兰表达式），计算表达式的值并进行输出。可以验证程序正确实现了构建表达式二叉树功能和表达式计算功能。

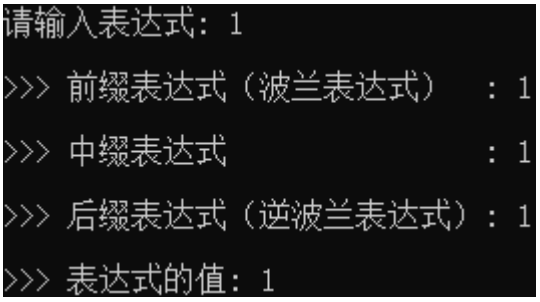

图 4.2.1 构建表达式二叉树功能和表达式计算功能测试（测试一）

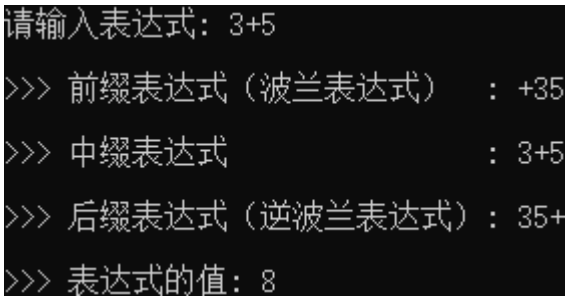

图 4.2.2 构建表达式二叉树功能和表达式计算功能测试（测试二）

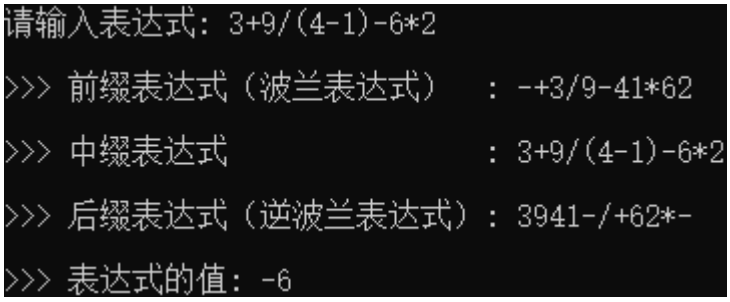

图 4.2.3 构建表达式二叉树功能和表达式计算功能测试（测试三）

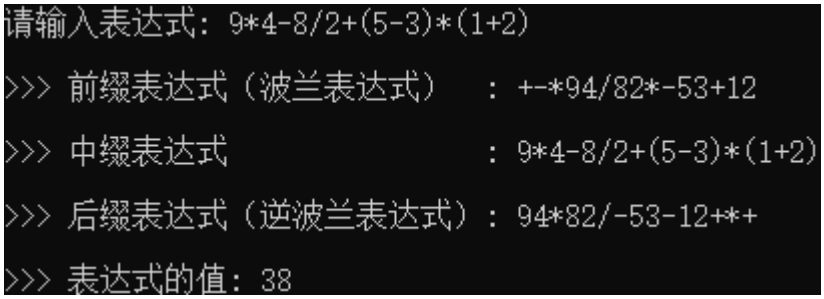

图 4.2.4 构建表达式二叉树功能和表达式计算功能测试（测试四）

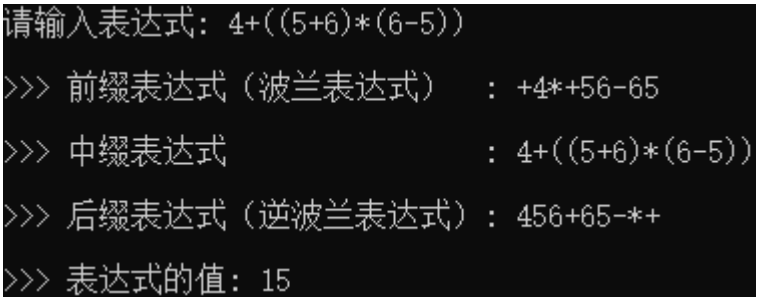

图 4.2.5 构建表达式二叉树功能和表达式计算功能测试（测试五）

5 集成开发环境与编译运行环境

Windows 系统：Windows 11 x64

Windows 集成开发环境：Microsoft Visual Studio 2022 (Release 模式)

Windows 编译运行环境：本项目适用于 x86 架构和 x64 架构

Linux 系统：CentOS 7 x64

Linux 编译命令：

g++ '/root/ 桌 面 /Share_Folder/expression_calculation.cpp' -o '/root/桌面/Share_Folder/expression_calculation' -lncurses

Linux 运行命令：

'/root/桌面/Share_Folder/expression_calculation'

图 5.1 Linux 环境程序运行示例

本项目使用条件编译解决 Windows 系统和 Linux 系统编译环境的差异，示例代码如下。

```
#ifdef _WIN32
#include <conio.h>
#elif __linux__
#include <ncurses.h>
#endif
```