**项目说明文档**

**数据结构课程设计**

**——表达式计算**

作 者 姓 名 刘淑仪

学 号 2251730

指 导 教 师 张 颖

学 院 专 业 软件学院 软件工程



二〇二三 年 十一 月 二十五 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

2 项目设计 2

2.1 数据结构设计 2

2.2 类结构设计 2

2.3 成员与操作设计 3

2.4 系统设计 5

3 功能设计与项目实现 6

3.1 总体设计功能的实现 6

3.1.1 总体系统实现思路 6

3.1.2 总体系统核心代码 6

3.1.3 总体功能实现示例 7

3.2 输入表达式功能的实现 7

3.2.1 输入表达式功能实现思路 7

3.2.2 输入表达式功能核心代码 8

3.2.3 输入功能实现示例 9

3.3 解析表达式与构建树功能的实现 9

3.3.1 解析表达式与构建树功能实现思路 9

3.3.2 解析表达式与构建树功能核心代码 9

3.3.3 解析表达式功能实现示例 10

3.4 计算表达式功能的实现 10

3.4.1 计算表达式功能实现思路 10

3.4.2 计算表达式功能核心代码 11

3.4.3 计算表达式功能实现示例 12

3.5 本项目涉及Expression Tree.h内容的主要实现 12

3.5.1 Expression Tree.h头文件中主要包含的类设计 12

3.5.2 Expression Tree.h核心代码 12

4 项目测试 13

4.1 功能测试 13

4.1.1 基本功能测试 13

4.1.2 括号优先级测试 13

4.1.3 单数字测试 14

4.1.4 复合表达式测试 14

4.1.5 长表达式测试 15

4.2 错误测试 16

4.2.1 输入验证错误 16

4.2.1.1 输入验证错误判断思路 16

4.2.1.3输入示例 16

4.3 Linux环境测试 18

5 集成开发环境与编译运行环境 18

# 项目分析

## 项目背景分析

在编程和计算机科学中，表达式求值是一项基本且关键的任务。它涉及将编程语言中的表达式（如算术运算）转换为可由计算机执行的形式。这一过程不仅是编程语言理解的核心，也是编译器设计的关键部分。表达式通常由操作数（如数字、变量）、运算符（如加、减、乘、除）和界定符（如括号）组成。在编程语言中，这些元素共同工作，形成复杂的计算和逻辑结构。

人类习惯使用中缀表达式，然而，对于计算机而言，直接处理这种表达式效率较低。计算机需要额外的信息来正确解析运算符的优先级和运算的顺序。后缀表达式，或逆波兰表达式，将运算符置于操作数之后，从而消除了中缀表达式中的括号和运算符优先级问题。这种格式更适合计算机处理，因为它简化了解析过程，并且可以直接使用栈结构进行求值。

为了有效地处理和转换表达式，本项目采用了表达式二叉树的概念。在这种树中，叶节点代表操作数，而内部节点代表运算符。这种结构不仅有助于表达式的转换（例如，将中缀表达式转换为后缀表达式），而且提供了一种直观的方式来执行表达式求值。项目的主要目标是接收一个中缀表达式，并将其转换为对应的后缀表达式（逆波兰表达式），进而计算其值。这一过程包括读取表达式、创建对应的表达式二叉树，并通过不同的遍历方法（前序、中序、后序遍历）来展示逆波兰式、中序表达式和波兰表达式，从而完成整个求值过程。

通过实现这个项目，可以深入理解表达式求值的机制，以及为什么需要将中缀表达式转换为后缀表达式。同时，这也是探索数据结构（特别是二叉树）和算法（特别是栈和树的遍历）在实际应用中的一个很好的实例。

## 项目功能分析

本项目的核心目标是实现一个表达式求值器，能够接收中缀表达式，将其转换为后缀表达式（逆波兰表达式），并进行计算。项目中涉及到的主要算法及其功能实现主要依赖于

1、表达式解析（中缀转后缀）：使用栈（Stack类）来处理运算符的优先级。遍历整个表达式，当遇到操作数时直接输出，遇到运算符时根据其与栈顶运算符的优先级进行相应处理。

2、表达式树构建：通过TreeNode类表示树节点。在构建表达式树时，遍历后缀表达式，使用栈存储暂时未处理的树节点。当遇到操作符时，从栈中弹出两个节点作为其左右子节点，然后将这个新构建的树节点压入栈中。

3、树的遍历：使用递归方法遍历表达式树。前序遍历先访问根节点，然后左子树，最后右子树；中序遍历先访问左子树，然后根节点，最后右子树；后序遍历先访问左子树，然后右子树，最后根节点。前序遍历用于输出波兰表达式，中序遍历用于输出中序表达式，后序遍历用于输出逆波兰表达式。

4、表达式求值：递归地遍历表达式树。当遇到操作数时直接返回其值；遇到操作符时，递归计算左右子节点的值，并根据操作符执行相应运算。

# 项目设计

## 数据结构设计

由上文项目功能分析来看，本项目的核心在于实现一个表达式求值器，这要求使用栈、树节点、表达式树和队列等数据结构。首先，栈在这里起到了关键作用，它不仅在中缀表达式转换为后缀表达式的过程中用于临时存储和排序运算符，以保证运算符的正确顺序和优先级处理，还在构建表达式树时用于暂存树节点。接着，树节点（TreeNode）构成了表达式树的基本单元，存储操作数或运算符，并通过链接来表示表达式中的结构关系。表达式树（MyBinaryTree）本身是项目的核心数据结构，它不仅能够表示经过转换的后缀表达式，还支持前序、中序和后序遍历，用于生成和输出不同形式的表达式，同时也是求值算法的执行基础。最后，队列（LinkedQueue）在树的层次遍历中发挥作用，确保按层次顺序访问树中的每个节点。总的来说，这些数据结构的综合运用不仅为中缀到后缀表达式的转换提供了强有力的支持，而且使得表达式的求值过程变得直观和高效，体现了在处理类似复杂编程问题时数据结构选择的重要性和有效性。

## 类结构设计

首先，创建了Stack和LinkedQueue类来实现栈和队列的基本操作。Stack类通过私有成员变量存储栈顶指针和栈的大小，公有成员则包括构造函数、析构函数、进栈（Push）、出栈（Pop）、获取栈顶元素（GetTop）等基本操作。接下来，在TreeNode类中，定义了表达式树的节点结构。每个节点存储一个值和指向其子节点的指针。这个类是构建表达式树的基础，它将在MyBinaryTree类中被进一步使用。

在MyBinaryTree类中，实现了表达式树的核心功能。这个类的私有成员包括指向树根的指针，而公有成员则提供了一系列操作，包括树的构造、析构、遍历（前序、中序、后序）、求值等方法。这些方法的实现是表达式求值逻辑的核心。最后，在ExpressionTree类中，集成了MyBinaryTree类，并扩展了与表达式求值直接相关的功能。这个类中的私有成员包括用于存储原始表达式和解析后表达式的数组，公有成员则包括解析表达式、构建表达式树、计算表达式值等方法。此外，还包括与用户交互相关的函数，如输入表达式、展示结果等。

## 成员与操作设计

Expression Tree.h内存储基本二叉树的实现，作为主CPP表达式树的基类

// 定义栈存储表达式

template<class Type>

struct StackNode {

Type data;

StackNode<Type>\* link;

StackNode(Type d = 0, StackNode<Type>\* l = nullptr) :data(d), link(l) {}

};

template<class Type>

class Stack {

private:

StackNode<Type>\* top;

int size;

public:

Stack() :top(NULL) ,size(0){} //构造函数

~Stack(); //析构函数

void Push(const Type& item); //进栈

Type Pop(); //出栈

Type GetTop(); //取栈顶元素

void MakeEmpty(); //置空栈

int IsEmpty()const { return top == NULL; } //判栈空否

void OverturnStack(); //翻转栈

int GetSize() const { return size; } // 获取栈的元素总数

};

// 定义队列便于进行层次序遍历

// 链表节点定义

template<typename Type>

class QueueNode {

public:

TreeNode<Type>\* data;

QueueNode<Type>\* next;

QueueNode(TreeNode<Type>\* node) : data(node), next(NULL) {}

};

// 链队定义

template<typename Type>

class LinkedQueue {

private:

QueueNode<Type>\* front; // 队列头指针

QueueNode<Type>\* rear; // 队列尾指针

public:

LinkedQueue() : front(NULL), rear(NULL) {}

~LinkedQueue() {

while (!isEmpty()) { dequeue(); }

}

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

void enqueue(TreeNode<Type>\* node) {

QueueNode<Type>\* newNode = new QueueNode<Type>(node);

if (rear == NULL) {

front = rear = newNode;

}

else {

rear->next = newNode;

rear = newNode;

}

}

TreeNode<Type>\* dequeue() {

if (isEmpty()) {

return NULL;

}

QueueNode<Type>\* temp = front;

TreeNode<Type>\* node = front->data;

front = front->next;

if (front == NULL) {

rear = NULL;

}

delete temp;

return node;

}

TreeNode<Type>\* getFront() const {

return isEmpty() ? NULL : front->data;

}

};

/\* 构建二叉树 \*/

// 孩子兄弟法定义结点

template<typename Type>

class TreeNode {

public:

Type value;

TreeNode\* firstChild, \* nextSibling;

TreeNode() : firstChild(NULL), nextSibling(NULL) {}

TreeNode(Type item, TreeNode<Type>\* l = NULL, TreeNode<Type>\* r = NULL) : value(item), firstChild(l), nextSibling(r) {}

template<typename T>

friend std::ostream& operator<<(std::ostream& os, const TreeNode<T>& node);

};

## 系统设计

在本项目中，代码被细致地分成了多个模块，每个模块承担着特定的职责，从而实现了表达式求值的完整功能。例如，Stack和LinkedQueue类专注于实现栈和队列的基本操作，为表达式的解析和树的层次遍历提供支持。TreeNode和MyBinaryTree类则负责构建和处理表达式树，这是求值逻辑的核心。最后，ExpressionTree类集成了所有这些功能，并提供了用户接口，使用户能够输入表达式、查看结果。

使用栈来存储操作符和树节点是处理表达式的关键。这种数据结构适合于中缀表达式转换为后缀表达式的需求，因为它能有效地处理运算符的优先级和顺序。同样，表达式树提供了一种直观的方式来组织和计算表达式，使得遍历和求值过程自然而直接。

在设计中，数据和方法被封装在各个类中，大大减少了全局状态的使用。这种封装不仅提高了代码的模块化程度，也增强了代码的可读性和可维护性。通过明确的接口和分离的职责，各个模块可以独立工作，同时又能协同完成复杂的表达式求值任务。总体而言，这种系统设计既确保了功能的完整性，又保持了代码的清晰和灵活性，为解决实际编程问题提供了一个有效的范例。

# 功能设计与项目实现

## 总体设计功能的实现

### 总体系统实现思路

首先实现了从中缀表达式到后缀表达式的转换逻辑。通过栈来处理运算符，确保了运算符的正确顺序和优先级。这个步骤是实现表达式求值的前提，确保了表达式可以被计算机正确理解和处理。根据解析后的后缀表达式，构建了表达式树。这一步骤使用栈来暂存节点，建立操作符和操作数的树结构。表达式树使得表达式的计算变得直观和可操作。

而后，实现了表达式树的前序、中序和后序遍历，这些遍历分别用于生成波兰表达式、中序表达式和逆波兰表达式。这个步骤是展示表达式在不同形式下的结构和求值过程的关键。实现了递归遍历表达式树以计算表达式的值。这是项目的最终目标，也是核心功能。

### 总体系统核心代码

int main()

{

ET Solution;

Solution.menu();

Solution.enterExpression();

Solution.operation();

}

// 输出操作界面

void ExpressionTree::operation()

{

std::cout << "\n波兰表达式：";

preOrder(root);

std::cout << std::endl;

std::cout << "\n中缀表达式：";

// inOrder(root);

for (int i = 0; expr[i] != '\0'; i++)

std::cout<<expr[i] << ' ';

std::cout << std::endl;

std::cout << "\n逆波兰表达式：";

postOrder(root);

std::cout << std::endl;

std::cout << "\n表达式的值为：";

int result = evaluateExpressionTree(root);

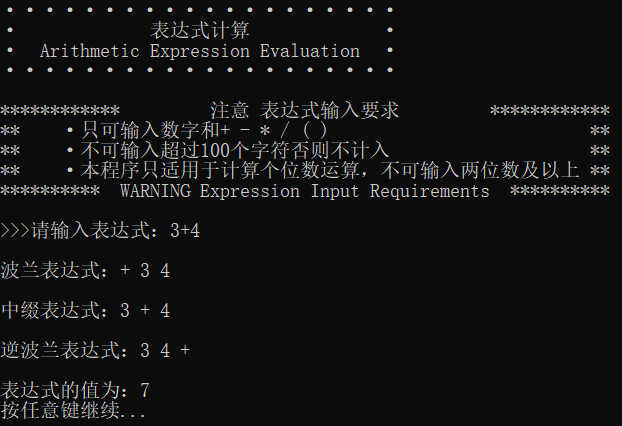
std::cout << result;

std::cout << std::endl;

Exit();

}

### 总体功能实现示例



## 输入表达式功能的实现

### 输入表达式功能实现思路

首先，函数通过一系列的 std::cout 语句向用户展示输入表达式的要求。这包括可接受的字符类型（数字和算术运算符），输入的字符长度限制，以及程序对输入表达式的特定限制（如只能处理个位数运算）。使用 std::cin.getline 方法接收用户输入的表达式。这里设置了最大字符数 MAX\_SIZE 来限制输入长度，防止潜在的缓冲区溢出问题。调用 parseExpression 方法将用户输入的中缀表达式转换为后缀表达式。这个过程涉及栈的使用，以处理运算符的优先级和括号。接着调用 buildExpressionTree 方法，基于转换后的后缀表达式构建表达式树。这个步骤是通过栈来临时存储节点，并按照后缀表达式的逻辑连接节点，构建出反映表达式逻辑的树结构。

### 输入表达式功能核心代码

// 输入表达式

void ExpressionTree::enterExpression()

{

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\* 注意 表达式输入要求 \*\*\*\*\*\*\*\*\*\*\*\*\n";

std::cout << "\*\* ·只可输入数字和+ - \* / ( ) \*\*\n";

std::cout << "\*\* ·不可输入超过100个字符否则不计入 \*\*\n";

std::cout << "\*\* ·本程序只适用于计算个位数运算，不可输入两位数及以上 \*\*\n";

std::cout << "\*\*\*\*\*\*\*\*\*\* WARNING Expression Input Requirements \*\*\*\*\*\*\*\*\*\*\n\n";

std::cout << ">>>请输入表达式：";

std::cin.getline(expr, MAX\_SIZE);

// 检查括号匹配和非法字符

int bracketCount = 0;

for (int i = 0; expr[i] != '\0'; i++) {

if (expr[i] == '(') {

bracketCount++;

}

else if (expr[i] == ')') {

bracketCount--;

}

if (!isOperand(expr[i]) && !isOperator(expr[i]) && expr[i] != '(' && expr[i] != ')') {

std::cerr << "错误：输入包含非法字符。\n";

return;

}

if (bracketCount < 0) {

std::cerr << "错误：括号不匹配。\n";

return;

}

}

if (bracketCount != 0) {

std::cerr << "错误：括号不匹配。\n";

return;

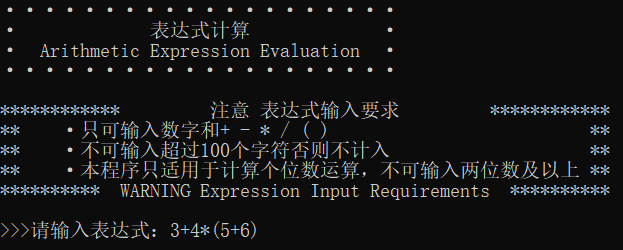
}

parseExpression();

buildExpressionTree();

}

### 输入功能实现示例



## 解析表达式与构建树功能的实现

### 3.3.1 解析表达式与构建树功能实现思路

在解析表达式中，使用一个栈 (opStack) 来存储操作符。定义一个索引 (parsedIndex) 用于跟踪解析后的后缀表达式数组 (parsedExpr)。然后对原始表达式 expr 进行遍历。对于每个字符 (token)，判断它是操作数、操作符还是括号。如果 token 是操作数（数字），直接将其添加到 parsedExpr。遇到左括号 ( 时，将其压入栈。遇到右括号 ) 时，不断弹出栈顶元素并添加到 parsedExpr，直到遇到左括号 (。遇到操作符时，根据其与栈顶操作符的优先级，决定是将其压入栈还是弹出栈顶元素到 parsedExpr。表达式遍历完成后，将栈中剩余的操作符依次弹出，添加到 parsedExpr。在 parsedExpr 末尾添加空字符 \0 表示字符串结束。

在构建表达式树中，先使用一个栈 (nodeStack) 来暂存表达式树的节点。然后再对解析后的后缀表达式 parsedExpr 进行遍历。如果 token 是操作数，为其创建一个树节点 (TreeNode<char>) 并压入栈。如果 token 是操作符，从栈中弹出两个节点作为其右、左子节点（注意后缀表达式的顺序）。为操作符创建一个新的树节点，并将其两个子节点设置为之前弹出的两个节点。将这个新创建的操作符节点压入栈。遍历完成后，栈顶元素即为整个表达式树的根节点，将其设置为 root。

### 3.3.2 解析表达式与构建树功能核心代码

// 解析表达式 中缀->后缀

void ExpressionTree::parseExpression()

{

Stack<char> opStack; // 存储操作符

int parsedIndex = 0; // 输出字符串的当前索引

for (int i = 0; expr[i] != '\0'; i++) {

char token = expr[i];

if (isOperand(token)) {

// 如果是操作数，直接输出到parsedExpr

parsedExpr[parsedIndex++] = token;

}

else if (token == '(') { opStack.Push(token); }

else if (token == ')') {

while (!opStack.IsEmpty() && opStack.GetTop() != '(') { parsedExpr[parsedIndex++] = opStack.Pop(); }

opStack.Pop(); // 弹出左括号

}

else if (isOperator(token)) {

// 遇到操作符

while (!opStack.IsEmpty() && precedence(opStack.GetTop()) >= precedence(token)) {

parsedExpr[parsedIndex++] = opStack.Pop();

}

opStack.Push(token);

}

}

// 将栈中剩余的操作符输出到parsedExpr

while (!opStack.IsEmpty()) {

parsedExpr[parsedIndex++] = opStack.Pop();

}

parsedExpr[parsedIndex] = '\0'; // 以空字符结束字符串

}

### 3.3.3 解析表达式功能实现示例

详见下实现示例。

## 计算表达式功能的实现

### 3.4.1 计算表达式功能实现思路

先检查传入的节点 (node) 是否为 nullptr。如果是，返回 0 或抛出异常。这是递归终止条件之一，确保了递归的安全性。如果当前节点是叶子节点（即操作数），直接返回其数值。这里需要将字符表示的数字转换为整数值。例如，字符 '2' 转换为整数 2，可以通过 node->value - '0' 实现。如果当前节点是操作符，递归地调用 evaluateExpressionTree 函数计算其左子树（node->firstChild）和右子树（node->nextSibling）的值。这是函数的核心递归逻辑，它遍历并计算表达式树的每个部分。使用 switch 语句根据节点的操作符（node->value）来执行相应的算术运算。如果遇到未知的操作符或运算过程中的错误（如除以零），函数抛出异常。

### 3.4.2 计算表达式功能核心代码

// 计算表达式

char ExpressionTree::evaluateExpressionTree(TreeNode<char>\* node)

{

// 检查节点是否为空

if (node == nullptr) {

return 0; // 或者抛出异常

}

// 如果是叶子节点（操作数），直接返回其数值

if (!isOperator(node->value)) {

return node->value - '0'; // 将字符转换为整数

}

// 递归计算左右子树

char left = evaluateExpressionTree(node->firstChild);

char right = evaluateExpressionTree(node->nextSibling);

// 根据操作符执行相应的运算

switch (node->value) {

case '+':

return left + right;

case '-':

return left - right;

case '\*':

return left \* right;

case '/':

// 注意处理除数为0的情况

if (right == 0) {

throw std::runtime\_error("Division by zero");

}

return left / right;

default:

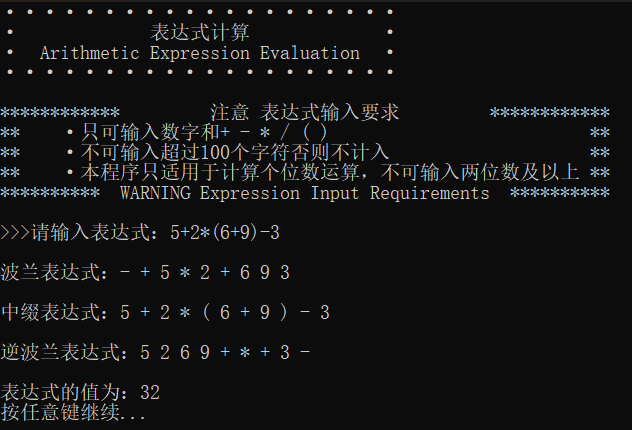
// 处理未知操作符

throw std::runtime\_error("Unknown operator");

}

}

### 3.4.3 计算表达式功能实现示例



## 本项目涉及Expression Tree.h内容的主要实现

### 3.5.1 Expression Tree.h头文件中主要包含的类设计

定义了二叉树结点TreeNode，每个节点包含值（value），指向第一个孩子（firstChild）和下一个兄弟（nextSibling）的指针。定义了LinkedQueue作为用于层次遍历二叉树的链队列，包含队列的头（front）和尾（rear）指针，以及相关的队列操作函数（如 enqueue、dequeue）。定义了二叉树的主体MyBinaryTree，表示二叉树的主体，包含根节点指针（root）以及一系列操作二叉树的函数，如 copy、destroy、isEmpty、getHeight、getSize 等。提供树的遍历方法，包括前序、中序、后序和层次序遍历。包含节点的查找、插入和删除操作。

### 3.5.2 Expression Tree.h核心代码

详见类设计。

# 项目测试

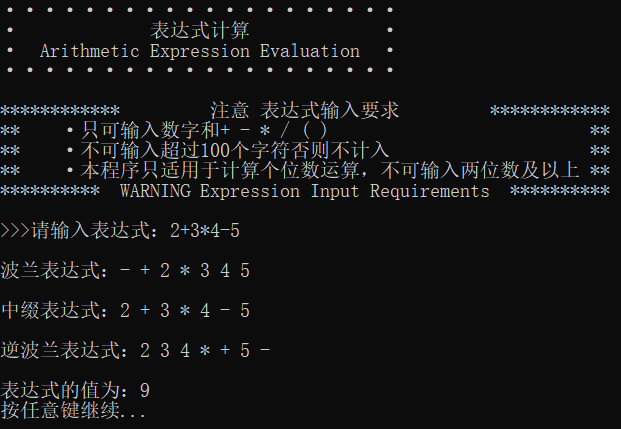
## 功能测试

### 基本功能测试

**测试用例：**2+3\*4-5

**预期结果：**9

**实验结果：**

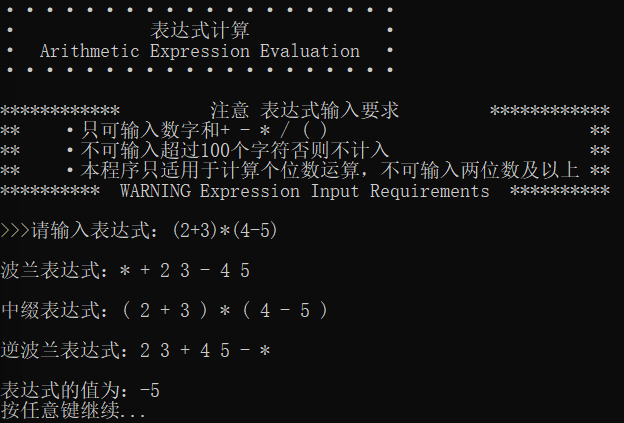


### 4.1.2 括号优先级测试

**测试用例：**（2+3）\*（4-5）

**预期结果：**-5

**实验结果：**

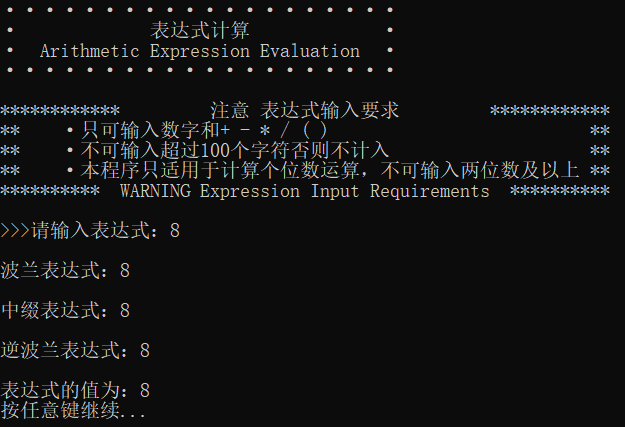


### 4.1.3 单数字测试

**测试用例：**8

**预期结果：**8

**实验结果：**

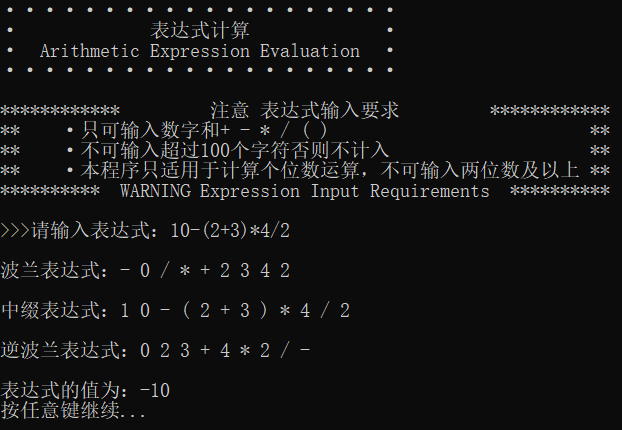


### 4.1.4 复合表达式测试

**测试用例：**10 - (2 + 3) \* 4 / 2

**预期结果：**-10

**实验结果：**

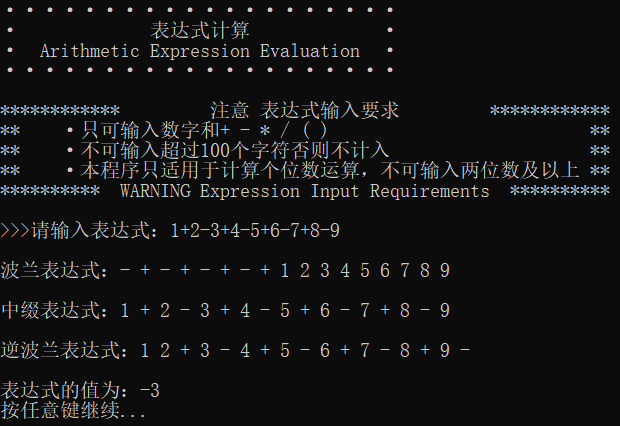


### 4.1.5 长表达式测试

**测试用例：**1 + 2 - 3 + 4 - 5 + 6 - 7 + 8 - 9

**预期结果：**-3

**实验结果：**



## 错误测试

### 输入验证错误

#### 4.2.1.1 输入验证错误判断思路

输入非法值（如负数、非数字字符、超出范围的数字）来测试程序对错误输入的处理能力。检查程序是否能妥善处理并提示用户重新输入。

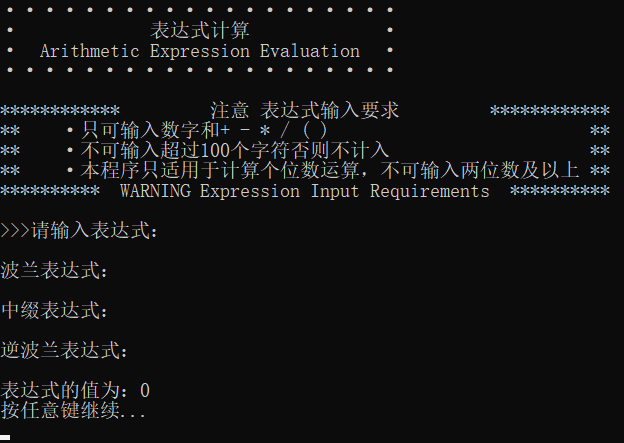
#### 4.2.1.3输入示例

**测试用例：**空字符串；

**用例说明：**空输入测试；

**预期结果：**无输出，表达式计算为0。

**实验结果：**

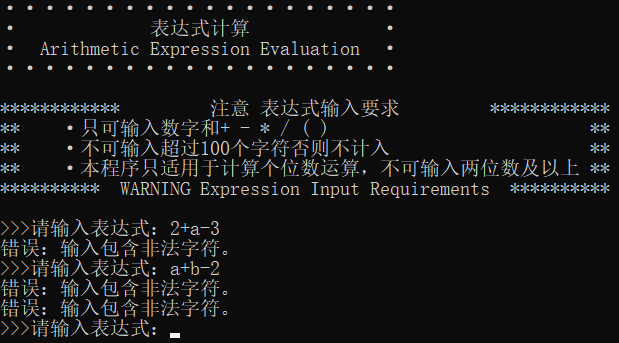


**测试用例：**2+a-3；

**用例说明：**无效字符测试；

**预期结果：**错误提示，非法字符。

**实验结果：**

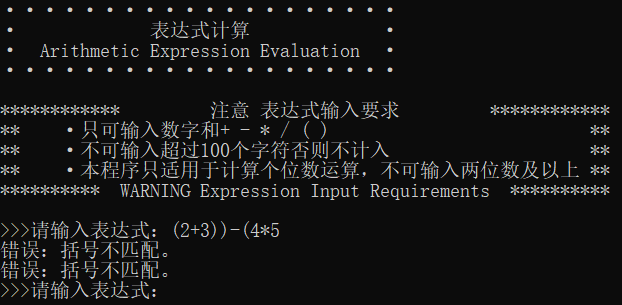


**测试用例：**(2 + 3)) - (4 \* 5；

**用例说明：**括号不匹配测试

**预期结果：**错误提示，括号不匹配。

**实验结果：**

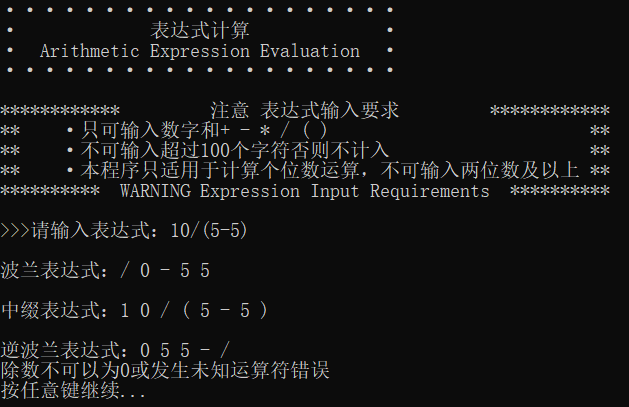


**测试用例：**10/（5-5）；

**用例说明：**除数为0测试

**预期结果：**运行时错误，除零异常。

**实验结果：**



## Linux环境测试

**编译命令：** g++ -std=c++11 '/mnt/hgfs/ShareFolder/ 07\_arithmetic\_expression\_evaluation' -o sevenExample

**运行命令：** ./sevenExample

**实验结果：**



# 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