**《数据结构》课程实践报告**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院、系 | 计算机学院 | | 年级专业 | 23软工 | 姓名 | 王天予 | 学号 | 2362401031 |
| 实验布置日期 | | 2024.10.8 | | 提交  日期 |  | | 成绩 |  |

课程实践实验3：栈的实现及应用

## **一、问题描述及要求**

### 1. 问题描述和要求

本实验的目标是使用C++设计并实现一个包含顺序栈和表达式处理器的计算器程序，主要功能包括：

1. **顺序栈类的实现**：构建自己的顺序栈类，提供基本的栈操作。
2. **表达式求值**：通过输入以“#”结尾的字符串形式的表达式，计算其结果。支持前缀、中缀、后缀表达式，并能自动识别表达式类型。支持加、减、乘、除、幂运算及括号。

示例：

* 中缀输入：3.5\*(20+4)-20/4#，输出：79
* 后缀输入：3.5 20 4 + \* 20 4 / - #，输出：79
* 前缀输入：- + \* 3 4 5 6 #，输出：11

## **二、概要设计**

### 1. 系统功能列表

**顺序栈类**：

- push(double value): 将值入栈。

- pop(): 从栈顶弹出值。

- top(): 返回栈顶元素。

- isEmpty(): 判断栈是否为空。

**表达式处理器类**：

- parseExpr(string expr): 解析并计算表达式，自动识别前、中、后缀表达式。

- tokenize(string expr): 将输入的表达式字符串分割成标记（tokens）。

- evalPrefix(vector<string> tokens): 计算前缀表达式的值。

- evalInfix(vector<string> tokens): 计算中缀表达式的值。

- evalPostfix(vector<string> tokens): 计算后缀表达式的值。

- operate(double opr1, double opr2, char op): 对两个操作数执行运算符操作。

- priority(char op): 返回运算符的优先级。

### 2. 程序运行界面设计

**交互式Shell**：

- 提供Shell界面，允许用户输入表达式。

- 根据输入表达式的格式，自动判断并调用相应的求值方法。

- 支持前缀、中缀、后缀表达式的输入，用户无需指定表达式类型。

## **三、详细设计**

### 1. 顺序栈类 ADT

#### ADT Stack

**数据对象**：

**数据关系**：

#### 基本操作：

1. **InitStack(Stack &S)**  
   初始化栈，使其成为一个空栈。
2. **DestroyStack(Stack &S)**  
   销毁栈，释放栈所占内存空间。
3. **Push(Stack &S, double value)**  
   栈顶插入元素value，若栈满，则返回错误。
4. **Pop(Stack &S, double &value)**  
   删除栈顶元素并将其赋给value，若栈空，则返回错误。
5. **GetTop(Stack S, double &value)**  
   获取栈顶元素并赋给value，不修改栈的内容。若栈空，则返回错误。
6. **StackEmpty(Stack S)**  
   若栈为空，则返回true；否则，返回false。

### 2. 表达式处理器类 ADT

#### ADT exprProcessor

**数据对象**：栈 等

**输入**: 表达式

**输出**: 表达式的值

#### 基本操作：

1. **parseExpr(exprProcessor &E, string expr)**  
   解析并计算表达式expr的值，自动识别表达式类型。
2. **tokenize(exprProcessor &E, string expr)**  
   将表达式字符串expr分割成标记（tokens）。
3. **evalPrefix(exprProcessor &E, vector tokens)**  
   计算前缀表达式并返回结果。
4. **evalInfix(exprProcessor &E, vector tokens)**  
   计算中缀表达式并返回结果。
5. **evalPostfix(exprProcessor &E, vector tokens)**  
   计算后缀表达式并返回结果。
6. **operate(exprProcessor &E, double opr1, double opr2, char op)**  
   计算opr1 op opr2的结果，并返回。
7. **priority(exprProcessor &E, char op)**  
   返回操作符op的优先级。

### 3. 设计思路

在顺序栈类的基础上，构建了表达式处理器类exprProcessor，能够处理不同类型的表达式。程序首先通过tokenize函数将输入的表达式字符串分割成标记，然后在parseExpr函数中自动识别表达式类型（前缀、中缀、后缀）。

* **前缀表达式**：从右到左扫描标记，遇到操作数则入栈，遇到操作符则从栈中弹出两个操作数进行计算。
* **中缀表达式**：使用操作数栈和操作符栈，根据运算符优先级和括号规则进行计算。
* **后缀表达式**：从左到右扫描标记，遇到操作数则入栈，遇到操作符则从栈中弹出两个操作数进行计算。

通过实现operate函数处理基本的运算操作，并通过priority函数确定运算符的优先级，确保表达式能够正确地被解析和计算。

## **四、实验结果**

### 测试数据及结果

**测试用例 1**：中缀表达式

* + 输入：3.5\*(20+4)-20/4#
  + 预期输出：79
  + 实际输出：79
  + 结果：通过

**测试用例 2**：中缀表达式，带负数

* + 输入：-3+4\*2-1#
  + 预期输出：4
  + 实际输出：4
  + 结果：通过

**测试用例 3**：后缀表达式

* + 输入：3.5 20 4 + \* 20 4 / - #
  + 预期输出：79
  + 实际输出：79
  + 结果：通过

**测试用例 4**：后缀表达式，带负数

* + 输入：5 3 - 2 \* #
  + 预期输出：4
  + 实际输出：4
  + 结果：通过

**测试用例 5**：前缀表达式

* + 输入：- + \* 3 4 5 6 #
  + 预期输出：11
  + 实际输出：11
  + 结果：通过

**测试用例 6**：自动识别表达式类型

* + 输入：\* + 3 4 5 #（前缀）
  + 输出：35
  + 输入：(3+4)\*5#（中缀）
  + 输出：35
  + 输入：3 4 + 5 \*#（后缀）
  + 输出：35
  + 结果：通过

## **五、实验分析与探讨**

### 1. 测试结果分析

1）**功能验证**：  
程序成功实现了对前缀、中缀、后缀表达式的解析和求值，且能够自动识别表达式类型。在多种输入格式下，计算结果均与预期一致，表明算法逻辑正确可靠。

2）**表达式格式的兼容性**：  
程序能够处理没有空格的中缀表达式，以及以空格分隔的前缀和后缀表达式。通过改进tokenize函数，增强了对不同输入格式的兼容性，使得用户输入更加灵活。

3）**错误处理**：  
程序增加了对非法字符、除零错误和表达式格式错误的检测，能够在遇到错误时给出相应的提示，提升了程序的健壮性。

### 2. 性能分析

1）**时间复杂度**：  
- **中缀表达式**：由于需要处理运算符的优先级和括号匹配，时间复杂度为O(n)，n为表达式长度。

- **前缀和后缀表达式**：单次遍历即可完成计算，时间复杂度为O(n)。

2）**空间复杂度**：  
使用栈来存储操作数和运算符，空间复杂度为O(n)。程序对内存的使用较为高效，适用于处理较长的表达式。

### 3. 实验设计和实现中的问题

1）**表达式类型的自动识别**  
描述：如何准确地自动识别输入的表达式是前缀、中缀还是后缀，是一个挑战。  
解决方法：通过检查表达式的第一个和最后一个标记是否为操作符，结合常见的表达式格式，推断出表达式类型。

2）**标记（Token）解析**  
描述：在处理没有空格的中缀表达式时，初始的tokenize函数无法正确解析数字和运算符的边界。  
解决方法：改进tokenize函数，增加对数字和运算符的细粒度解析，支持多位数字、小数和负数的识别。

3）**负数和减号的区分**  
描述：在解析负数时，需要区分减号作为运算符和负号作为数字的一部分。  
解决方法：在tokenize函数中，结合上下文判断“-”符号是作为负号还是减号，并相应地解析。

## **六、总结与反思**

在本次实验中，我们成功地实现了一个支持前缀、中缀、后缀表达式求值的计算器程序，并能够自动识别表达式类型。这一过程加深了我们对栈这种数据结构的理解，以及表达式求值算法的掌握。

### 1. **顺序栈的实现与应用**

通过手动实现顺序栈，巩固了对栈的基本操作和特性的理解。在表达式求值过程中，栈被广泛应用于存储操作数和运算符，其先进后出的特性非常适合这种应用场景。

### 2. **表达式解析与求值**

在支持多种表达式格式的情况下，如何准确地解析和计算表达式，是本次实验的核心。我们通过改进标记解析方法，能够处理复杂的输入格式。同时，通过自动识别表达式类型，提升了程序的智能化程度。

### 3. **程序的健壮性与用户体验**

在程序设计中，考虑到用户可能的各种输入情况，增加了对错误的检测和提示。这不仅提高了程序的健壮性，也提升了用户体验。

### 4. **实验收获与展望**

本次实验强化了我们对基本数据结构和算法的理解，并让我们深入体验了手动实现和应用顺序栈的过程。通过不使用STL，强迫自己掌握了底层的操作实现，这无疑对加深数据结构基础具有重要意义。与此同时，我们还认识到在程序设计中，充分考虑异常输入的处理能力、对用户操作的响应等都是构建健壮性程序的重要因素。

**附录：源代码**

1. **实验环境**：

编译环境：TDM-GCC 4.9.2 64bit release

命令行参数：-static-libgcc -std=c++11

2、**目录结构**

（1）Problem.cpp

#include <iostream>

#include <cstring>

#include <cctype>

#include <cstdlib>

#include <vector>

#include <cmath>

#include <string>

using namespace std;

template <typename T>

class Stack {

private:

T \*data;

int topIndex;

int capacity;

public:

Stack(int size) : topIndex(-1), capacity(size) {

data = new T[size];

}

~Stack() {

delete[] data;

}

void push(T value) {

if (topIndex == capacity - 1) {

cout << "Stack overflow" << endl;

exit(1);

}

data[++topIndex] = value;

}

T pop() {

if (topIndex == -1) {

cout << "Stack underflow" << endl;

exit(1);

}

return data[topIndex--];

}

T top() const {

if (topIndex == -1) {

cout << "Stack is empty" << endl;

exit(1);

}

return data[topIndex];

}

bool isEmpty() const {

return topIndex == -1;

}

};

class exprProcessor{

public:

double parseExpr(string expr);

private:

vector<string> tokenize(string expr);

int priority(char op);

double operate(double opr1, double opr2, char op);

double evalPrefix(vector<string> tokens);

double evalInfix(vector<string> tokens);

double evalPostfix(vector<string> tokens);

bool isOperator(const string& token);

};

vector<string> exprProcessor::tokenize(string expr) {

vector<string> tokens;

string token;

for (size\_t i = 0; i < expr.length(); i++) {

char ch = expr[i];

if (isspace(ch)) {

continue;

}

if (isdigit(ch) || ch == '.') {

token += ch;

while (i + 1 < expr.length() && (isdigit(expr[i + 1]) || expr[i + 1] == '.')) {

token += expr[++i];

}

tokens.push\_back(token);

token.clear();

} else if (ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '^') {

if (ch == '-' && (tokens.empty() || isOperator(tokens.back()) || tokens.back() == "(")) {

token += ch;

while (i + 1 < expr.length() && (isdigit(expr[i + 1]) || expr[i + 1] == '.')) {

token += expr[++i];

}

tokens.push\_back(token);

token.clear();

} else {

tokens.push\_back(string(1, ch));

}

} else if (ch == '(' || ch == ')' || ch == '#') {

tokens.push\_back(string(1, ch));

} else {

cout << "Invalid character: " << ch << endl;

exit(1);

}

}

return tokens;

}

bool exprProcessor::isOperator(const string& token) {

return token == "+" || token == "-" || token == "\*" || token == "/" || token == "^";

}

int exprProcessor::priority(char op) {

switch (op) {

case '^': return 3;

case '\*':

case '/': return 2;

case '+':

case '-': return 1;

default: return 0;

}

}

double exprProcessor::operate(double opr1, double opr2, char op) {

switch (op) {

case '+': return opr1 + opr2;

case '-': return opr1 - opr2;

case '\*': return opr1 \* opr2;

case '/':

if (opr2 == 0) {

cout << "Division by zero error" << endl;

exit(1);

}

return opr1 / opr2;

case '^':

return pow(opr1, opr2);

default:

cout << "Unknown operator: " << op << endl;

exit(1);

}

}

double exprProcessor::evalPostfix(vector<string> tokens) {

Stack<double> stack(tokens.size());

for (const string& token : tokens) {

if (token == "#") {

break;

} else if (isdigit(token[0]) || (token[0] == '-' && token.length() > 1 && isdigit(token[1]))) {

stack.push(atof(token.c\_str()));

} else if (isOperator(token)) {

if (stack.isEmpty()) {

cout << "Invalid expression" << endl;

exit(1);

}

double opr2 = stack.pop();

if (stack.isEmpty()) {

cout << "Invalid expression" << endl;

exit(1);

}

double opr1 = stack.pop();

double result = operate(opr1, opr2, token[0]);

stack.push(result);

} else {

cout << "Invalid token: " << token << endl;

exit(1);

}

}

if (!stack.isEmpty()) {

double result = stack.pop();

if (!stack.isEmpty()) {

cout << "Invalid expression" << endl;

exit(1);

}

return result;

} else {

cout << "Invalid expression" << endl;

exit(1);

}

}

double exprProcessor::evalPrefix(vector<string> tokens) {

Stack<double> stack(tokens.size());

for (int i = tokens.size() - 1; i >= 0; i--) {

const string& token = tokens[i];

if (token == "#") {

continue;

} else if (isdigit(token[0]) || (token[0] == '-' && token.length() > 1 && isdigit(token[1]))) {

stack.push(atof(token.c\_str()));

} else if (isOperator(token)) {

if (stack.isEmpty()) {

cout << "Invalid expression" << endl;

exit(1);

}

double opr1 = stack.pop();

if (stack.isEmpty()) {

cout << "Invalid expression" << endl;

exit(1);

}

double opr2 = stack.pop();

double result = operate(opr1, opr2, token[0]);

stack.push(result);

} else {

cout << "Invalid token: " << token << endl;

exit(1);

}

}

if (!stack.isEmpty()) {

double result = stack.pop();

if (!stack.isEmpty()) {

cout << "Invalid expression" << endl;

exit(1);

}

return result;

} else {

cout << "Invalid expression" << endl;

exit(1);

}

}

double exprProcessor::evalInfix(vector<string> tokens) {

Stack<double> values(tokens.size());

Stack<char> operators(tokens.size());

for (size\_t i = 0; i < tokens.size(); ++i) {

const string& token = tokens[i];

if (token == "#") {

break;

} else if (isdigit(token[0]) || (token[0] == '-' && token.length() > 1 && isdigit(token[1]))) {

values.push(atof(token.c\_str()));

} else if (token == "(") {

operators.push('(');

} else if (token == ")") {

while (!operators.isEmpty() && operators.top() != '(') {

char op = operators.pop();

double val2 = values.pop();

double val1 = values.pop();

double result = operate(val1, val2, op);

values.push(result);

}

if (!operators.isEmpty() && operators.top() == '(') {

operators.pop();

} else {

cout << "Invalid expression" << endl;

exit(1);

}

} else if (isOperator(token)) {

char currOp = token[0];

while (!operators.isEmpty() && priority(operators.top()) >= priority(currOp)) {

char op = operators.pop();

double val2 = values.pop();

double val1 = values.pop();

double result = operate(val1, val2, op);

values.push(result);

}

operators.push(currOp);

} else {

cout << "Invalid token: " << token << endl;

exit(1);

}

}

while (!operators.isEmpty()) {

char op = operators.pop();

double val2 = values.pop();

double val1 = values.pop();

double result = operate(val1, val2, op);

values.push(result);

}

if (!values.isEmpty()) {

double result = values.pop();

if (!values.isEmpty()) {

cout << "Invalid expression" << endl;

exit(1);

}

return result;

} else {

cout << "Invalid expression" << endl;

exit(1);

}

}

double exprProcessor::parseExpr(string expr) {

vector<string> tokens = tokenize(expr);

if (tokens.empty()) {

cout << "Empty expression" << endl;

exit(1);

}

if (!tokens.empty() && tokens.back() == "#") {

tokens.pop\_back();

}

if (tokens.empty()) {

cout << "Empty expression after removing '#'" << endl;

exit(1);

}

bool isOperatorFirst = isOperator(tokens[0]);

bool isOperatorLast = isOperator(tokens.back());

if (isOperatorFirst && !isOperatorLast) {

return evalPrefix(tokens);

} else if (!isOperatorFirst && isOperatorLast) {

return evalPostfix(tokens);

} else {

return evalInfix(tokens);

}

}

int main() {

exprProcessor processor;

char expression[100];

while (true) {

cout << "Enter expression (end with '#' or 'exit' to quit): ";

cin.getline(expression, 100);

if (strcmp(expression, "exit") == 0) break;

if (!strchr(expression, '#')) {

cout << "Missing '#'.\n";

continue;

}

cout << "Evaluation result: " << processor.parseExpr(string(expression)) << endl;

}

return 0;

}