**《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**

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. **中缀表达式求值**：通过输入以“#”结尾的字符串形式的中缀表达式，计算其结果。支持加、减、乘、除及括号。
3. **后缀表达式求值**：通过输入空格分隔、以“#”结尾的后缀表达式，计算其结果。

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

## **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。

- peek(): 返回栈顶元素。

- isEmpty(): 判断栈是否为空。

**表达式处理器类**：

- evaluateInfix(const char \*expression): 求值中缀表达式。

- evaluatePostfix(const char \*expression): 求值后缀表达式。

- precedence(char op): 返回运算符的优先级。

- applyOperation(double a, double b, char op): 对两个数执行运算符操作。

- readNumber(const char \*&expr): 从表达式中读取数字。

### 2. 程序运行界面设计

**交互式Shell**：

- 提供Shell界面，允许用户输入中缀或后缀表达式。

- 根据输入表达式格式（是否含空格）自动判断使用中缀或后缀求值方法。

## **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 ExpressionProcessor

**数据对象**：栈 等

**输入**: 中缀或后缀表达式

**输出**: 表达式的值

#### 基本操作：

1. **EvaluateInfix(ExpressionProcessor &E, const char expression[])**  
   计算中缀表达式并返回结果。
2. **EvaluatePostfix(ExpressionProcessor &E, const char expression[])**  
   计算后缀表达式并返回结果。
3. **Precedence(ExpressionProcessor &E, char op)**  
   返回操作符op的优先级。
4. **ApplyOperation(ExpressionProcessor &E, double a, double b, char op)**  
   计算a op b的结果，并返回。
5. **ReadNumber(ExpressionProcessor &E, const char &expr[])**  
   从表达式字符串中读取一个数字，并将指针移到数字后的第一个字符。

### 3. 设计思路

顺序栈类采用数组来存储数据，操作数和运算符分别使用两个栈来进行中缀表达式求值。后缀表达式则通过扫描元素并对操作数入栈、操作符执行运算的方式求解。Shell功能允许用户动态输入表达式并获取计算结果。

## **四、实验结果**

### 测试数据及结果

1. **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
2. **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
3. **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
4. **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

## **五、实验分析与探讨**

### 1. 测试结果分析

1）**功能验证**：  
 所有基本功能测试均通过，包括中缀和后缀表达式的解析和求值。程序能够正确处理加减乘除运算及括号优先级。在输入多种格式的表达式后，程序输出的计算结果与手工计算结果一致，表明核心算法逻辑正确。

2）**表达式格式的兼容性**：  
 程序能够兼容中缀和后缀两种表达式格式，并能识别以空格分隔的后缀表达式及带括号的中缀表达式。这一特性提升了程序的使用灵活性，使得用户可以选择不同的表达方式进行计算。

### 2. 性能分析

1）**时间复杂度**：  
 后缀表达式的求值算法需要处理整个表达式，并根据操作符的优先级动态调整操作数栈和运算符栈。由于每个字符最多只入栈出栈一次，该算法的时间复杂度为O(n)，其中n是表达式的长度。后缀表达式求值仅需单次遍历，时间复杂度同样为O(n)，因此程序能够高效地处理较长的表达式。

2）**空间复杂度**：  
 程序的空间消耗主要受操作数和操作符栈的影响，栈空间复杂度为O(n)。考虑到栈的动态增长不会超过表达式长度的一半，整体空间复杂度较低，适合在内存资源受限的环境中运行。对于非常长的表达式，程序的内存消耗仍然在可接受范围内。

### 3. 实验设计和实现中的问题

1）**运算符优先级处理**  
 描述：在最初设计时，中缀表达式的运算符优先级实现较为复杂，需要频繁比较栈顶运算符的优先级并执行出栈操作，导致代码可读性较差。  
解决方法：通过实现precedence函数，简化了优先级判断逻辑，使得运算符的入栈和出栈操作更加清晰。这提高了代码的可维护性，也减少了运算过程中的误判情况。

2）**浮点数解析**  
 描述：程序需要支持包含小数的浮点运算。在最初实现中，由于浮点数的解析方式不够完善，导致部分小数运算出现错误。  
解决方法：通过增加readNumber函数，对小数点和小数位进行识别和处理。这样改进后，程序能够正确解析和计算带有小数的表达式，确保了浮点运算的准确性。

3）**表达式结束符检测**  
 描述：在Shell中设计交互式表达式输入时，如何有效识别表达式结束符（#）是一个关键问题。没有结束符检测会导致程序等待用户输入的状态持续到非预期的时间。  
 解决方法：使用#作为表达式的结束标志，并在读取到结束符后立即开始处理。这样不仅简化了输入检测的逻辑，也提升了程序的响应速度，使用户可以更快地得到计算结果。

## **六、总结与反思**

在本次实验中，通过手动实现顺序栈和表达式处理器，我们更深入地理解了栈的操作机制和表达式求值的原理。这项实验不仅巩固了对栈这种数据结构的认识，也帮助我们掌握了中缀表达式转换及求值的核心算法。

### 1. **顺序栈的实现与使用**

顺序栈在操作上要求我们关注栈的容量控制和栈顶指针的变化。在实验中，我们通过一个固定大小的数组来实现栈的存储，并实现了栈的基本操作（如入栈、出栈、读取栈顶等）。该实现过程中遇到的挑战主要在于防止栈的溢出和下溢，这也使我们更深入理解了数据结构的边界条件控制。通过这些手动操作的练习，加深了对栈的LIFO特性和实现细节的理解。

### 2. **表达式处理器的实现与改进**

在表达式处理器部分，我们分别实现了对中缀和后缀表达式的求值。对于中缀表达式，我们运用了两个栈来分别存储操作数和运算符，并且在运算符入栈时根据其优先级进行判断。对于后缀表达式，我们则是采用了直接扫描操作数入栈、遇到运算符弹出计算的方式。

这次实验中的中缀表达式求值算法让我们认识了操作符优先级和括号嵌套的处理细节，并学习了如何在不借助标准库的情况下手动解析和计算表达式。此过程还暴露出了一些潜在的改进点，例如当前版本对输入格式较为依赖，对非法字符和错误输入的处理还不够健全。这为我们后续在异常处理和鲁棒性设计方面提供了提升的空间。

### 3. **实验收获与反思**

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>

using namespace std;

class Stack {

private:

double \*data;

int top;

int capacity;

public:

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

data = new double[size];

}

~Stack() {

delete[] data;

}

void push(double value) {

if (top == capacity - 1) {

cout << "Stack overflow" << endl;

exit(1);

}

data[++top] = value;

}

double pop() {

if (top == -1) {

cout << "Stack underflow" << endl;

exit(1);

}

return data[top--];

}

double peek() const {

if (top == -1) {

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

exit(1);

}

return data[top];

}

bool isEmpty() const {

return top == -1;

}

};

class ExpressionProcessor {

public:

double evaluateInfix(const char \*expression);

double evaluatePostfix(const char \*expression);

private:

int precedence(char op);

double applyOperation(double a, double b, char op);

double readNumber(const char \*&expr, bool isNegative);

};

int ExpressionProcessor::precedence(char op) {

if (op == '+' || op == '-') return 1;

if (op == '\*' || op == '/') return 2;

return 0;

}

double ExpressionProcessor::applyOperation(double a, double b, char op) {

switch (op) {

case '+': return a + b;

case '-': return a - b;

case '\*': return a \* b;

case '/': return a / b;

}

return 0;

}

double ExpressionProcessor::readNumber(const char \*&expr, bool isNegative) {

double result = 0;

double fraction = 1;

bool isFraction = false;

while (isdigit(\*expr) || \*expr == '.') {

if (\*expr == '.') {

isFraction = true;

expr++;

continue;

}

if (isFraction) {

fraction \*= 0.1;

result += (\*expr - '0') \* fraction;

} else {

result = result \* 10 + (\*expr - '0');

}

expr++;

}

return isNegative ? -result : result;

}

double ExpressionProcessor::evaluateInfix(const char \*expression) {

Stack values(100);

Stack ops(100);

bool expectNegative = true;

while (\*expression != '#') {

if (\*expression == ' ') {

expression++;

continue;

}

if (isdigit(\*expression) || (\*expression == '-' && expectNegative)) {

bool isNegative = (\*expression == '-');

if (isNegative) expression++;

values.push(readNumber(expression, isNegative));

expectNegative = false;

} else if (\*expression == '(') {

ops.push(\*expression++);

expectNegative = true;

} else if (\*expression == ')') {

while (!ops.isEmpty() && ops.peek() != '(') {

double val2 = values.pop();

double val1 = values.pop();

char op = (char)ops.pop();

values.push(applyOperation(val1, val2, op));

}

ops.pop();

expression++;

expectNegative = false;

} else {

while (!ops.isEmpty() && precedence(ops.peek()) >= precedence(\*expression)) {

double val2 = values.pop();

double val1 = values.pop();

char op = (char)ops.pop();

values.push(applyOperation(val1, val2, op));

}

ops.push(\*expression++);

expectNegative = true;

}

}

while (!ops.isEmpty()) {

double val2 = values.pop();

double val1 = values.pop();

char op = (char)ops.pop();

values.push(applyOperation(val1, val2, op));

}

return values.pop();

}

double ExpressionProcessor::evaluatePostfix(const char \*expression) {

Stack stack(100);

while (\*expression != '#') {

if (\*expression == ' ') {

expression++;

continue;

}

if (isdigit(\*expression) || (\*expression == '-' && isdigit(\*(expression + 1)))) {

stack.push(readNumber(expression, \*expression == '-'));

if (\*expression == '-') expression++;

} else {

double val2 = stack.pop();

double val1 = stack.pop();

stack.push(applyOperation(val1, val2, \*expression++));

}

}

return stack.pop();

}

void runShell() {

ExpressionProcessor 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;

}

if (strchr(expression, ' ')) {

cout << "Postfix evaluation result: " << processor.evaluatePostfix(expression) << endl;

} else {

cout << "Infix evaluation result: " << processor.evaluateInfix(expression) << endl;

}

}

}

int main() {

runShell();

return 0;

}