简易计算器软件的研究与制作

奉贤中学 高二(5)班 毛佳俊

语言：C++

1、软件目的

本软件主要为了实现简单四则代数式的运算。

2 、实现方法

首先，最开始的思路来源于日常的手动计算。

例如`1-2\*(3+2)`这个代数式，我们会先计算括号中的3+2,然后再计算1-2*5，然后遵从先乘除后加减的运算规则，依次计算。

总的步骤就是：

```
1-2*(3+2)
=1-2*5
=1-10
=-9
```

其次我们考虑括号的匹配问题。

例如`(()())`这个括号组合，我们可以发现，可以通过栈这个数据结构来进行括号的匹配，遍历到左括号就将其下标压入栈中，如果遍历到了右括号，那么直接弹出栈顶与其匹配。

但是，我在第一个版本的实现中并没有直接利用到栈，而是参考了栈的思路，遍历到左括号就将左括号的下标记录在变量`L`中，然后后续如果再次遍历到左括号就只需要更新变量`L`，这样变量`L`就相当于是栈顶的元素。如果遍历到右括号就直接与变量`L`记录的左括号进行匹配。

然后，我们就可以得到如下思路。

遍历字符串`formula`，优先进行括号的匹配，一匹配到一对合法的括号(左右括号下标分别记录在`L`,`R`变量中)就将括号内的代数式计算出答案，然后将原先字符串`formula[L,R]`替换为计算出的答案再继续计算。

对于+-*/四则运算，我们参考同样的思路，优先遍历*/如果遍历到了*/运算符，我们再从运算符的位置向两边搜索，搜索出两个待运算的数字（两个搜索边界分别记录在`L`,`R`变量中），然后同样将字符串`formula[L,R]`替换成计算结果再继续计算。

综上，我们总是通过从一个完整的代数式中选取一个单符号运算式(指的是只有一个运算符的代数式例如1+1等)计算出结果并替换原字符串再进行计算。这样，我们就把一个多符号运算式的计算问题简化成了一个个单符号运算问题，利用分而治之的编程思想来解决整个问题。下面是一个我利用递归的实现。

```cpp
//C++
#include <algorithm>
#include <iostream>
#include <math.h>
#include <regex>
#include <string>

using std::cin;
using std::cout;
using std::endl;
using std::string;

std::string replace(std::string str, const std::string &pattern, const
std::string &to)
{
    std::regex r(pattern);
    return std::regex_replace(str, r, to);
}

bool isNum(char c)
{
    return ('0' <= c && c <= '9');
}

bool isOperator(char c)
{
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

double SingleOperatorCalc(string formula)
{
    int len = formula.length();
    char operator_ = '+';
    int index = 0;
    double num1, num2;
    // 找到运算符
    for (unsigned int i = 0; i < len; i++) {
        char t = formula[i];
        if (isOperator(t)) {
            operator_ = t;
            index = i;
            break;
        }
    }

    num1 = atof(formula.substr(0, index).c_str());
    num2 = atof(formula.substr(index + 1, len).c_str());
    switch (operator_) {
    case '+':
        return num1 + num2;
        break;
```

```
    case '-':
        return num1 - num2;
        break;
    case '*':
        return num1 * num2;
        break;
    case '/':
        return num1 / num2;
        break;

    default:
        break;
    }
    return 0;
}

double _MutipleOperatorCalc(string formula)
{
    // cout << formula << endl;
    int len = formula.length();
    for (unsigned int i = 0; i < len; i++) {
        char t = formula[i];
        if (t == '*' || t == '/') {
            int L = i, R = i;
            while (L - 1 >= 0 && !isOperator(formula[L - 1])) {
                L--;
            }
            while (R + 1 <= len - 1 && !isOperator(formula[R + 1])) {
                R++;
            }
            int tLen = R - L + 1;                                    // 子代
数式的长度
            double sAns = SingleOperatorCalc(formula.substr(L, tLen)); // 子代
数式的答案
            return _MutipleOperatorCalc(formula.replace(L, tLen,
std::to_string(sAns)));
        }
    }

    for (unsigned int i = 0; i < len; i++) {
        char t = formula[i];
        if (isOperator(t) && i != 0) {
            int index = i;
            int L = 0, R = index;
            while (R + 1 <= len - 1 && !isOperator(formula[R + 1])) {
                R++;
            }
            int tLen = R - L + 1;                                    // 子代
数式的长度
            double sAns = SingleOperatorCalc(formula.substr(L, tLen)); // 子代
数式的答案
```

```
            return _MutipleOperatorCalc(formula.replace(L, tLen,
std::to_string(sAns)));
        }
    }
    return atof(formula.c_str());
}

string __MutipleOperatorCalc(string formula)
{
    cout << "=" << formula << endl;
    int len = formula.length();
    int L = 0;
    int R = len;
    for (int i = 0; i < len; i++) {
        if (formula[i] == '(') {
            // 记录左括号
            L = i;
        } else if (formula[i] == ')') {
            // 搜索到右括号就结束
            R = i;
            int tLen = R - L - 1;
 // 子代数式的长度
            double sAns = _MutipleOperatorCalc(formula.substr(L + 1, tLen));
// 子代数式的答案
            return __MutipleOperatorCalc(formula.replace(L, tLen + 2,
std::to_string(sAns)));
            break;
        }
    }
    return formula;
}

double MutipleOperatorCalc(string formula)
{
    formula = replace(formula, "--", "+");
    formula = __MutipleOperatorCalc(formula);
    return _MutipleOperatorCalc(formula);
}
```

只需要调用`MutipleOperatorCalc(string formula);`就可以得到运算结果。其中`formula`是字符串类型的代数式。

在提供的源码中，该函数包含在`calc.hpp`头文件下，只需要在主函数所在的源文件中包含这个头文件，并对其调用即可。

```
//example
//calc.hpp与主函数所在的源文件处于统一目录下
#include "calc.hpp"

int main()
{
    string formula;
    cin >> formula;
    cout << MutipleOperatorCalc(formula);
    return 0;
}
```

对部分源码进行解释

```
double __MutipleOperatorCalc(string formula);    //匹配括号，将括号内的内容交给_MutipleOperatorCalc处理
double _MutipleOperatorCalc(string formula);     //匹配四则运算，四则运算交给SingleOperatorCalc处理
double SingleOperatorCalc(string formula);       //单符号运算式计算
```

上述可以看出整个程序的基本架构。

3、算法缺点

- 过多的重复搜索
- 直接处理字符串

这些缺点使得算法效率并不是特别理想

4、算法优化

结合对原先算法缺点的分析，我们可以通过将遍历出来的数字和符号保存在容器内，直接对容器的元素进行操作，这样只需要一次遍历就可以解析好代数式，这样就可以避免直接操作字符串带来的不必要的内存和时间的消耗。

同时我们完全可以直接利用栈的性质来代替原先的递归(递归本质上也是栈的应用)

算法优化思路(非原创)

遍历一次字符串，将解析出来的数字和运算符保存在`operator_stack`(运算符栈)和`num_stack`(数字栈)中。

维护`operator_stack`使得出栈的时候保证栈顶的运算符是具有较高运算优先级的，即维护`operator_stack`中的符号的优先级从栈顶开始是严格递减的。

当即将入栈的运算符优先级小于当前栈顶的运算符时，**弹出`num_stack`中两个数字，以及`operator_stack`栈顶的运算符，提前进行计算，并将计算结果重新压入`num_stack`中。**(操作1)

遍历一次后进行出栈操作，重复执行**操作1**，直到`operator_stack`中的所有符号都弹出，此时`num_stack`中的元素就是运算的答案。

关于括号的匹配，遍历到左括号的时候只需要直接压入，但是当遍历到右括号的时候需要持续执行**操作1**直到有左括号从栈顶弹出。（注意，需要把左括号的优先级设为最低，使得后续的符号可以成功入栈）

下面是基于这个思路自己写的代码实现

```
#pragma once

#include <chrono>
#include <iostream>
#include <stack>
#include <string>
using namespace std;
/*
 *利用栈实现简单代数式的计算
 */

stack<char> operator_stack; // 运算符栈
stack<int> num_stack;       //  数字栈

bool isNum(char c)
{
    return ('0' <= c && c <= '9');
}

bool isOperator(char c)
{

    return (c == '+' || c == '-' || c == '*' || c == '/' || c == '(' || c == ')');
}
int getPriority(char c)
{
    switch (c) {
    case '(':
        return 0;
    case '+':
    case '-':
        return 1;
    case '*':
    case '/':
        return 2;
    default:
        return -1;
    }
}
int calc(int num1, char oper, int num2)
{
    // cout << num1 << oper << num2 << endl;
    switch (oper) {
    case '+':
        return num1 + num2;
```

```
    case '-':
        return num1 - num2;
    case '*':
        return num1 * num2;
    case '/':
        return num1 / num2;
    }
    return -1;
}
void stack_calc()
{
    int num1 = num_stack.empty() ? 0 : num_stack.top();
    num_stack.pop();
    int num2 = num_stack.empty() ? 0 : num_stack.top();
    num_stack.pop();
    char oper = operator_stack.top();
    operator_stack.pop();
    num_stack.push(calc(num2, oper, num1));
}
int mutiOperatorCalc(string formula)
{
    int len = formula.length();
    int ans;
    /*
     *入栈
     TODO 维护operator_stack使得出栈的时候保证栈顶的运算符是具有较高运算优先级的.
     TODO 当即将入栈的运算符优先级小于当前栈顶的运算符时,
     TODO 压出num_stack中两个数字, 以及operator_stack栈顶的运算符,提前进行计算,
     TODO 并将计算结果重新压入num_stack中
    */
    for (int i = 0; i < len; ++i) {
        char c = formula[i];
        if (isNum(c)) {
            string num = "";
            num += c;
            // 向后搜索完全整个数字
            while (i + 1 < len && isNum(formula[i + 1]))
                num += formula[++i];
            // 将数字压入数字栈
            num_stack.push(atoi(num.c_str()));
        } else if (isOperator(c)) { // 处理运算符
                                    // 检查即将入栈的运算符的优先级是否*不小于*栈顶的运算符优先级
            if (!operator_stack.empty()) {
            }
            if (c == '(')
                operator_stack.push(c);
            else if (c == ')') {
                // cout << "123";
```

```
                while (!operator_stack.empty() && operator_stack.top() != '(')
{
                    stack_calc();
                }
                operator_stack.pop();
            } else {
                while (!operator_stack.empty() && getPriority(c) <=
getPriority(operator_stack.top())) {
                    stack_calc();
                }
                operator_stack.push(c);
            }
        }
    }
    /*
     *出栈
     */
    // cout << "出栈" << endl;
    while (!operator_stack.empty()) {
        stack_calc();
    }

    return num_stack.top();
}
```

优化后的算法解决了字符串处理和重复搜索的问题，同时也运用到了前文提到过的利用栈匹配括号的思路，利用栈的特性将栈的特点发挥的淋漓尽致。

5、测试

```
void test(int formula_len)
{
    // 这是个随机生成代数式的函数并计算
    srand(time(0));
    char a[] = {'+', '-', '*', '/'};
    string formula = std::to_string(rand() % 10000);
    for (int j = 0; j < formula_len; j++) {
        formula += (a[rand() % 4] + std::to_string(rand() % 10000));
    }
    printf("%s=%f\n", formula.c_str(), MutipleOperatorCalc(formula));
    // cout <<endl;
}
```

可以利用这个函数自动生成四则运算式（不含括号）进行测试。

测试数据

测试数据：1000个1000个运算符的代数式

- 测试函数生成代数式　　time cost:0.067s
- 优化前的算法(含生成代数式的时间)　　time cost:7.451s
- 优化后的算法(含生成代数式的时间)　　time cost:0.359s

经过测试和计算，优化后的代码比优化前速度上快了大约 **25倍** 。

6、拓展研究方向

可以进一步研究：

- 高精度运算
- 科学运算(添加更多的运算方式，如幂，取模，三角等)
    - 由于科学运算符不是单字符的，例如 `sin()` 可以结合 括号匹配 和 数字搜索 的思路，解析整个 单符号科学运算式 来实现。
- UI可视化交互界面(MFC、QT)

7、项目结构

报告.md 为研究报告源文件，markdown格式

优化 文件夹下为优化后的代码

原先思路 文件夹下的为原先思路的代码

两个文件夹下

- **main.cpp** 为主函数所在文件
- **main.exe** 为可执行文件，可以通过用户输入来计算
- **calc.hpp** 算法核心实现文件，以 .hpp 的文件类型保存，便于移植调用。

8、总结

此次研究旨在解决对四则代数式的 解析 和 计算 ，初期以 分而治之 的核心思想，利用 递归 和 字符串操作 实现。但是这种处理方法效率较低，后期经过反思，利用了 栈 的性质来优化了算法。使得运算效率大幅提升。

奉贤中学　高二(　5　)班　毛佳俊

2023年2月18日