苏 州 大 学 实 验 报 告

院、系	计算机学院	年级专业	19 计算机类	姓名	张昊	学号	1927405160
课程名称	数据结构课程实践					成绩	
指导教师	孔芳	同组实验者	无		实验日期	2020 年 10 月 9 日	

实 验 名 称 中缀表达式求解

一、实验目的

通过本次实验要达到如下目的：

1. 熟悉栈“后进先出”的操作特性；
2. 掌握栈基本操作的实现；
3. 掌握应用栈来进行中缀表达式、括号处理的方法。

二、问题描述及要求

中缀表达式求解

中缀表达式是我们熟悉的表达式形式。为了能正确表示运算的先后顺序，中缀表达式中难免要出现括号。假设我们的表达式中只允许有圆括号。读入一个浮点数为操作数的中缀表达式后，对该表达式进行运算。要求中缀表达式以一个字符串的形式读入，可含有加、减、乘、除运算符和左、右括号，并假设该表达式以“#”作为输入结束符。

如输入“3.5*(20+4)-1#”，则程序运行结果应为 83。要求可单步显示输入序列和栈的变化过程。并考虑算法的健壮性，当表达式错误时，要给出错误原因的提示。

三、概要设计

1. 问题简析

中缀表达式是常见的一种表达式形式，其求值问题也是一个比较常见的问题，在编译程序中，这项任务通常由编译器来处理。对于中缀表达式，不能简单地按照“先左后右”的次序执行表达式中的运算符。运算符执行次序的规则（即运算优先级），一部分决定于事先约定的惯例（比如乘除优先于加减），另一部分则决定于括号。这就需要利用栈的 LIFO 的特性来处理好这种关系。

2. 程序结构设计思路

为了程序的封装性，将用于计算中缀表达式的值的各个函数和一些变量封装成 InfixExpression 类，方便使用者调用。InfixExpression 类提供一个带有一个参数的构造函数，便于使用者初始化一个中缀表达式；提供公有方法 calculate 来输出计算结果；当表达式错误时，通过抛出运行时异常来提示用户错误的原因。

由于表达式的计算需要多个部分的构成，在一个函数中执行过多的处理时不现实的，故在 InfixExpression 类中设计多个属性为私有的辅助函数，来辅助表达式的解析和计算过程。

对于需求“可单步显示输入序列和栈的变化过程”，在 InfixExpression 类中提供公有静

态方法 showDetail，询问用户是否需要显示输入序列和栈的变化过程。

以上为 InfixExpression 类的设计，放置于 util 文件夹中。

受 Python 等解释性语言的启发，程序设计为可以逐行计算表达式的形式，逐行计算包括两种情况：第一种是用户逐行输入（从标准输入）中缀表达式，程序进行计算；第二种是用户把要计算的多行表达式写在一个文件中，通过传入命令行参数的形式来执行这些表达式的计算。在文件 main.cpp 中设计两个交互函数，在主函数中根据参数的不同调用相应的函数来实现。

程序的算法基于栈结构来展开，故复用了顺序栈这一个现有数据结构。考虑到本次实验是基于这种数据结构的应用， 故其实现细节在报告中不再赘述。 有关顺序栈的实现，可见源代码文件夹 src 中 util/container 文件夹下的代码。

3. 程序界面设计

受 Python 等解释性语言的启发，程序设计为可以逐行计算表达式的形式，界面和使用方法也于 Python 的 shell 类似。

（1）交互执行模式

直接运行程序，首先询问用户是否需要单步显示输入序列和栈的变化过程。之后输出等待提示符“>>> ”，等待用户输入表达式（输入的表达式不要求以“#”作为输入结束符）。用户输入表达式后，若表达式正确，输出变化过程（如果需要的话）和结果；否则输出错误提示信息。用户输入“exit”后程序结束。如图为不显示变化过程的运行界面。

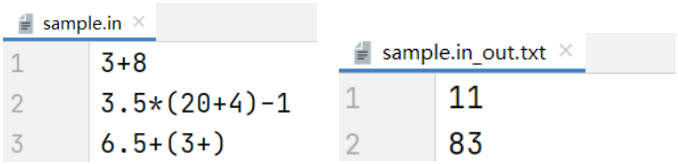
```
Infix Expression Calculator (Oct. 06 2020) By Holger Zhang
Type "exit" to exit the shell.
  < '~' means minus '-' >

* Show the changing process of the stack and the expression? [Y/n] n
>>> 3+5
8
>>> 3.5*(20+4)-1#
83
>>> 3.5/3+2/0
Unexpected line in <stdin> : 3.5/3+2/0
[When computing] Division by 0 error
>>> 6.5+(3+)
Unexpected line in <stdin> : 6.5+(3+)
[Popping operands] Invalid expression is given: missing operand(s)
>>> exit
```

（2）文件执行模式

运行程序时传入要执行的文件的位置参数，此时默认不显示输入序列和栈的变化过程。程序逐行执行文件中的表达式并保存到文件名+“_out.txt”文件中。如果遇到错误的表达式即输出错误的表达式以及错误信息，不再继续执行剩余的内容。如图为输入示例文件“sample.in”的运行界面。

```
Unexpected line in "sample.in" : 6.5+(3+)
[Popping operands] Invalid expression is given: missing operand(s)
```

图为输入示例文件“sample.in”和“sample.in_out.txt”的内容。



四、详细设计

对一个包含运算符+、-、*、/和小括号()的中缀表达式进行求值，需要进行以下操作。

1 操作数和运算符的保存

在中缀表达式的处理问题中，输入的表达式可分解为多个单元（操作数、操作符）并通过迭代依次扫描处理，但得到运算符后还需要等待其右操作数以及下一个运算符的，比较优先级后才能做出判断并计算。总的来说，这一过程中的各运算符的计算滞后于扫描到各运算符的进度，需要等到一些必要的信息较为完整时，才能计算得出结果。在这种情况下，可以**利用栈结构来充当数据的缓冲区**。

具体来说，使用两个栈分别保存遇到的操作数和运算符（为方便类成员函数的使用，将其定义为类的私用成员变量），需要保存的时候压栈，需要读取前面保存下来的内容则出栈。

2 运算符优先级的确定

与后缀表达式（又称逆波兰式）不同，后缀表达式的运算顺序由给定表达式中运算符的出现次序决定，而中缀表达式为表示运算的优先级，需要用到括号来调整运算次序。

对于事先约定的惯例，乘除的优先级高于加减；同一级别的运算符，**先出现的优先级高于后出现的**。对于括号，**优先处理括号包围的子表达式**：遇到左括号时，认为其优先级最高，将其压入栈；遇到右括号时，括号包围的子表达式已经入栈，则去寻找与其匹配的左括号并运算包括的表达式。比较两运算符的算法见下 C++代码：

```cpp
int InfixExpression::operatorCompare(char old_op, char new_op) noexcept {
    // 优先处理')'，并且不将其压入栈
    switch (old_op) {
        case '(':   // 左括号已经入栈，等待右括号
            return -1;
        case '*':   // 仅次于'('; 优先级比其他运算符都高 (包括同级运算符)
        case '/':
            if (new_op == '(') return -1;
            return 1;
         case '+':  // 优先级仅高于同级运算符;比其他运算符都低(不包括同级运算符)
         case '~':  // '~' 表示减法('-')
             if (new_op == '+' || new_op == '~') return 1;
             return -1;
```

```
    }
    exit(1);
}
```

3 表达式预处理

3.1 括号匹配检查

对于一个包含括号的表达式来说，括号的匹配是保证其运算顺序的前提条件。在表达式计算之前检查括号的匹配可以提早发现不符合要求的表达式，减少不必要的运算，同时防止对于不合法的表达式输出意料之外的结果。

采用一个栈结构，**将栈的 push、pop 操作分别与左、右括号相对应，必然与由 n 对括号组成的合法表达式彼此对应**。按照这一理解，只需扫描一趟表达式，即可在线性的时间内，判定其中的括号是否匹配。算法可由如下代码实现：

```
void InfixExpression::matchBracketCheck(const std::string &expression) {
    Stack<char> stack;
    for (char p : expression) {
        if (p == '(') {
            stack.push(p);
            continue;
        }
        if (p == ')' && (stack.empty() || stack.pop() != '(')) {
            throw std::runtime_error{"miss '('"};
         }
    }
    if (!stack.empty()) { throw std::runtime_error{"miss ')'"}; }
}
```

3.2 减号和负号的区分

在中缀表达式中，'-'的含义有二：减法运算符和负号，且这两个含义的操作数数量不同，减号需要两个操作数，而负号需要一个操作数，不能对他们做统一的处理。对于'-'的区分，主要判别方法为（表达式预先去除了空格和'#'字符）：
1）若'-'的前一个字符为'('，则必定为负号；
2）若'-'的前一个字符为')'或者数字，则必定为减号；
3）若前面一个字符为其他运算符，则必定是负号；
4）若前面没有字符，即该字符为表达式的第一个字符，则必定是负号；
5）若后面不是数字，视作减号。
也就是说在这种情况下，'-'是作为减号使用的：**前一个字符为')'或者数字，或者其后紧跟的不是数字**。

通过以上规则，**如果判断当前的'-'是作为减号使用的，则将其修改为'~'，即将'~'作为减法运算符，而原'-'仅保留负号的含义**。算法可以由如下代码实现：

```
void InfixExpression::determineMinus(std::string &expression, int pos) {
```

```
    if (pos != 0 && pos != expression.length() - 1
         && expression[pos] == '-') {
        if (expression[pos-1] == ')' || isdigit(expression[pos-1])
            || !isdigit(expression[pos+1])) {
            // '-'是减号：前一个字符为')'或者数字，或者其后紧跟的不是数字
            expression[pos] = '~';
        }
    }
    if (pos == 0 && expression[pos] == '-' &&
        !isdigit(expression[pos + 1])) {   // 首个字符后紧跟不是数字
        expression[pos] = '~';
    }
    if (pos == expression.length() - 1 &&
        expression[pos] == '-') {  // 其后紧跟的('\000')不是数字
        expression[pos] = '~';
    }
}
```

3.3 分离各个操作数和运算符

考虑使用一至多个空格来区别各运算符与操作数。在移除了用户输入时输入的空格和行尾标识符'#'，并消除了'-'的二义性后（即将'~'作为减法运算符，'-'作为负号），可以顺序查找各运算符'+'，'~'（减法运算符），'*'，'/'，'('，')'，并在其左右添加空格。这样，运算符和操作数就以空格分开了。在正式的计算过程中，利用 C++的字符串流即可很容易地将它们分开。

4 计算表达式

在计算之前进行表达式计算器的初始化，将操作数栈和运算符栈清空，以保证计算过程不会受到其他表达式的结果的干扰。

首先是对给定表达式的预处理，使用上面第三节提到的预处理方法，依次进行括号匹配检查、用'-'和'~'区分负号和减号、检查不合法字符、用空格分离各个操作数和运算符。之后，利用字符串流，将预处理好的表达式按空格分割，并循环读入表达式的每一部分。

```
matchBracketCheck(expression); // 括号匹配检查
std::istringstream scanner(parseExpression(expression));  // 预处理
std::string nextString;
while (scanner >> nextString) {
    // ......
}
```

其中 parseExpression 函数进行表达式的预处理，包括用'-'和'~'区分负号和减号、检查不合法字符、用空格分离各个操作数和运算符：

```
std::string InfixExpression::parseExpression(const std::string &expression){
```

```
    std::string new_expression(expression);
    for (int i = 0; i < new_expression.length(); ++i) {
        if (!isValidChar(new_expression[i])) {  // 检查不合法字符
            throw std::runtime_error{"Invalid character"};
        }
        determineMinus(new_expression, i);  // 区分负号和减号
    }
    splitOperatorsOperands(new_expression);  // 分离各个操作数和运算符
    return new_expression;
}
```

得到表达式的一个部分后，判断是否为**操作数**（判断方法很简单，遍历这一部分的字符，看是否全为数字 0-9 或'-'或'.'即可）：若为操作数则**将其转换为浮点数并压入操作数栈**，继续读入表达式；否则为运算符，进行运算符的处理。

运算符的处理流程为**边处理边计算**：当前运算符如果为**第一个**运算符，或者其优先级**大于**运算符栈顶的运算符优先级，则将**当前运算符压入栈**，继续读入表达式；若当前运算符的优先级**小于**运算符栈顶的运算符优先级，则从运算符栈弹出一个运算符，并从操作数栈分别弹出右操作数和左操作数，**计算结果压入操作数栈，并将当前运算符压入运算符栈**。另外，当得到右括号时，**优先处理括号包围的表达式**，即不断令运算符栈和操作数栈出栈，并执行计算，直到遇到左括号为止。

```
char op = nextString[0];
if (op == ')') {
    while (!operators.empty() && operators.top() != '(') {
        calculateNext();
    }
    getOperator(op);  // 删除栈中的 '('
    continue;
}
if (!operators.empty() && operatorCompare(operators.top(), op) > 0) {
    calculateNext();
}
saveOperator(op);
```

其中 `calculateNext` 封装了常用的出栈-计算-压栈的代码。

当遇到表达式结尾时，表达式中复杂的优先级和表达式嵌套关系已经处理好了，接下来要进行的就是**简化后的表达式处理**，并且各部分已经入栈。具体做法为：不断使运算符栈出栈，并从操作数栈中弹出相应的左右操作数，执行计算并压入操作数栈，直到**运算符栈为空**。

最后，从操作数栈读取**栈顶元素**作为整个表达式计算结果。程序中栈的压入与弹出操作封装为辅助函数，便于错误处理和流程的输出；整个函数只负责逐个读取表达式的组成部分并简单地调用各个封装好的功能。

五、实验结果测试

考虑到算法的健壮性，在表达式预处理的函数和涉及栈操作的函数中加入了异常的捕获与抛出，程序通过抛出运行时异常来指出输入的表达式不正确的地方，由主调方捕获并输出。程序计算出错如图所示：

```
Infix Expression Calculator (Oct. 06 2020) By Holger Zhang
Type "exit" to exit the shell.
  < '~' means minus '-' >

* Show the changing process of the stack and the expression? [Y/n] n
>>> 1+2*(5+(3-9)*2
Unexpected line in <stdin> : 1+2*(5+(3-9)*2
[Checking the bracket matching of the expression] Invalid bracket matches of the expression:
 miss ')'
>>> 1+2*5+(3-9)*2)
Unexpected line in <stdin> : 1+2*5+(3-9)*2)
[Checking the bracket matching of the expression] Invalid bracket matches of the expression:
 miss '('
>>> 3+5^2
Unexpected line in <stdin> : 3+5^2
[Parsing the expression] Invalid expression is given: contain invalid character
>>> 2 + 3 -
Unexpected line in <stdin> : 2 + 3 -
[Popping operands] Invalid expression is given: missing operand(s)
>>> 2(-3)
Unexpected line in <stdin> : 2(-3)
[Getting result] Invalid expression is given: missing operator(s)
```

同时，算法可以单步显示输入序列和栈的变化过程，可通过参数调整。

程序正常运行，不显示输入序列和栈的变化过程，如图所示：

```
Infix Expression Calculator (Oct. 06 2020) By Holger Zhang
Type "exit" to exit the shell.
  < '~' means minus '-' >

* Show the changing process of the stack and the expression? [Y/n] n
>>> 3.5*(20+4)-1
83
>>> 2.5*+3
Unexpected line in <stdin> : 2.5*+3
[Popping operands] Invalid expression is given: missing operand(s)
>>> exit
```

程序正常运行，显示输入序列和栈的变化过程，如图所示：

```
Infix Expression Calculator (Oct. 06 2020) By Holger Zhang
Type "exit" to exit the shell.
  < '~' means minus '-' >

* Show the changing process of the stack and the expression? [Y/n] y
>>> 3.5*(20+4)-1
 $ Checking the bracket matching of the expression...
 $ Parsing the expression...
 * Got parsed expression: 3.5 *  ( 20 + 4 )  ~ 1
 * Now get "3.5" from the expression.
◁ Push number 3.5 into the operands stack.
 * Now get "*" from the expression.
◁ Push operator '*' into operators stack.
 * Now get "(" from the expression.
◁ Push operator '(' into operators stack.
 * Now get "20" from the expression.
◁ Push number 20 into the operands stack.
 * Now get "+" from the expression.
◁ Push operator '+' into operators stack.
 * Now get "4" from the expression.
◁ Push number 4 into the operands stack.
 * Now get ")" from the expression.
 $ The sub expression in brackets will be calculated in advance.
▷ Operator '+' pop out of operators stack.
▷ Right operand 4 pop out of the operands stack.
▷ Left operand 20 pop out of the operands stack.
◁ Push number 24 into the operands stack.
▷ Operator '(' pop out of operators stack.
 $ The calculation of sub expression finished.
 * Now get "~" from the expression.
▷ Operator '*' pop out of operators stack.
▷ Right operand 24 pop out of the operands stack.
▷ Left operand 3.5 pop out of the operands stack.
◁ Push number 84 into the operands stack.
◁ Push operator '~' into operators stack.
 * Now get "1" from the expression.
◁ Push number 1 into the operands stack.
 * The expression reaches '#' or EOL.
▷ Operator '~' pop out of operators stack.
▷ Right operand 1 pop out of the operands stack.
▷ Left operand 84 pop out of the operands stack.
◁ Push number 83 into the operands stack.
▷ Get the result 83 from the operands stack.
83
>>> 2.5*+3
 $ Checking the bracket matching of the expression...
 $ Parsing the expression...
 * Got parsed expression: 2.5 *  + 3
 * Now get "2.5" from the expression.
◁ Push number 2.5 into the operands stack.
 * Now get "*" from the expression.
◁ Push operator '*' into operators stack.
 * Now get "+" from the expression.
▷ Operator '*' pop out of operators stack.
▷ Right operand 2.5 pop out of the operands stack.
Unexpected line in <stdin> : 2.5*+3
[Popping operands] Invalid expression is given: missing operand(s)
>>> exit
```

六、附录

1. **源代码路径**：C++源代码位于附件中 src 目录下。
2. **文件编码**：UTF-8；**行分隔符**：LF（\n）。
3. 文件输入样例见 src 目录下的 sample.in，其中最后一行为错误输入样例。
4. **实验环境**：Linux 操作系统，g++编译器，基于 cmake 构建；在 CLion 集成开发环境中调试运行通过。手动编译运行方法为（在 Linux/Unix shell 中）：

```
$ mkdir build  # pwd: src/
$ cd build
$ cmake ..
$ make
$ ./InfixExpressionCalculator [path-to-file]
```