哈尔滨工业大学计算学部

实验报告

课程名称：数据结构与算法

课程类型：专业核心基础课（必修）

实验项目：线性结构及其应用

实验题目：算术表达式求值（算术计算器）

实验日期：2023/9/20

班级：2203101

学号：2022113573

姓名：张宇杰

设计成绩	报告成绩	指导老师
		张岩

一、实验目的

表达式求值是实现程序设计语言的基本问题之一，也是栈的应用的一个典型例子。一个算术表达式是由操作数（operand）、运算符（operator）和界限符（delimiter）组成的。假设操作数是正整数，运算符只含加减乘除等四种二元运算符，界限符有左右括号和表达式起始、结束符“#”，如：#（7+15）*（23-28/4）#。引入表达式起始、结束符是为了方便。设计一个程序，演示算术表达式求值的过程。

二、实验要求及实验环境

实验要求：

1. 从文本文件输入任意一个语法正确的（中缀）表达式，显示并保存该表达式。
2. 利用栈结构，把（中缀）表达式转换成后缀表达式，并以适当的方式展示栈的状态变化过程和所得到的后缀表达式。
3. 利用栈结构，对后缀表达式进行求值，并以适当的方式展示栈的状态变化过程和最终结果。
4. 选做：将操作数类型扩充到实数、扩充运算符集合，并引入变量操作数，来完成表达式求值。
5. 选做：设计和实现结合 2 和 3 的“算法优先法”。

实验环境：

Windows 11, g++

三、设计思想（本程序中的用到的所有数据类型的定义，主程序的流程图及各程序模块之间的调用关系、核心算法的主要步骤）

1. 逻辑设计

枚举类型 TermType

```
1. enum TermType
2. {
3.     NUM = 0,
```

```
    OPERATOR
};
```

Term 类

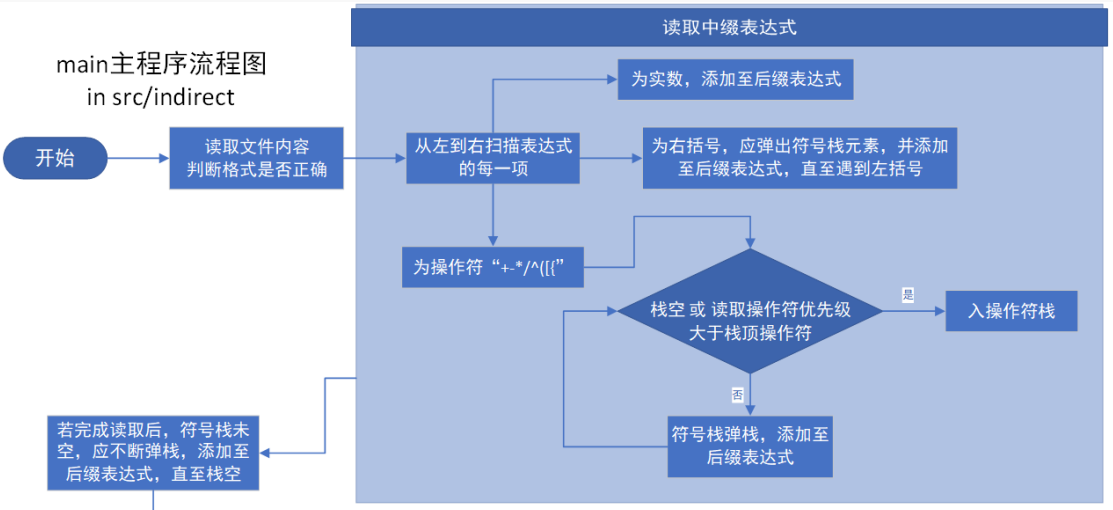
```
class Term
{
public:
    TermType type;
    string str;
    Term(TermType type, string str): type(type), str(str) {}
    Term(TermType type, char ch) : type(type), str(1, ch) {}
};
```

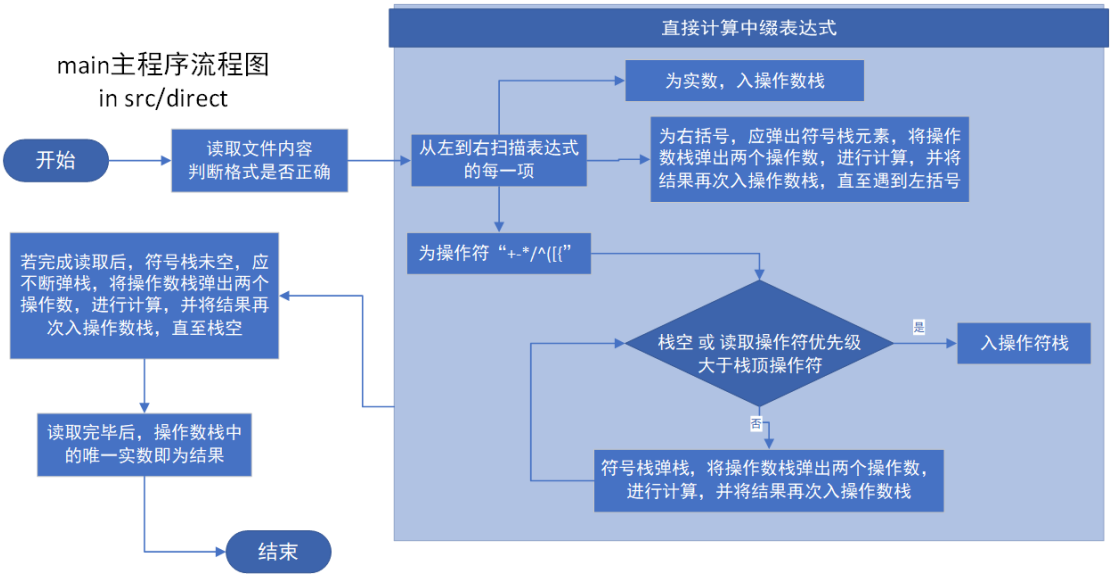
Term 类用于临时存放后缀表达式中的项，TermType 记录项类型
在直接计算(direct)的程序中没有使用，仅在间接计算(indirect)中出现

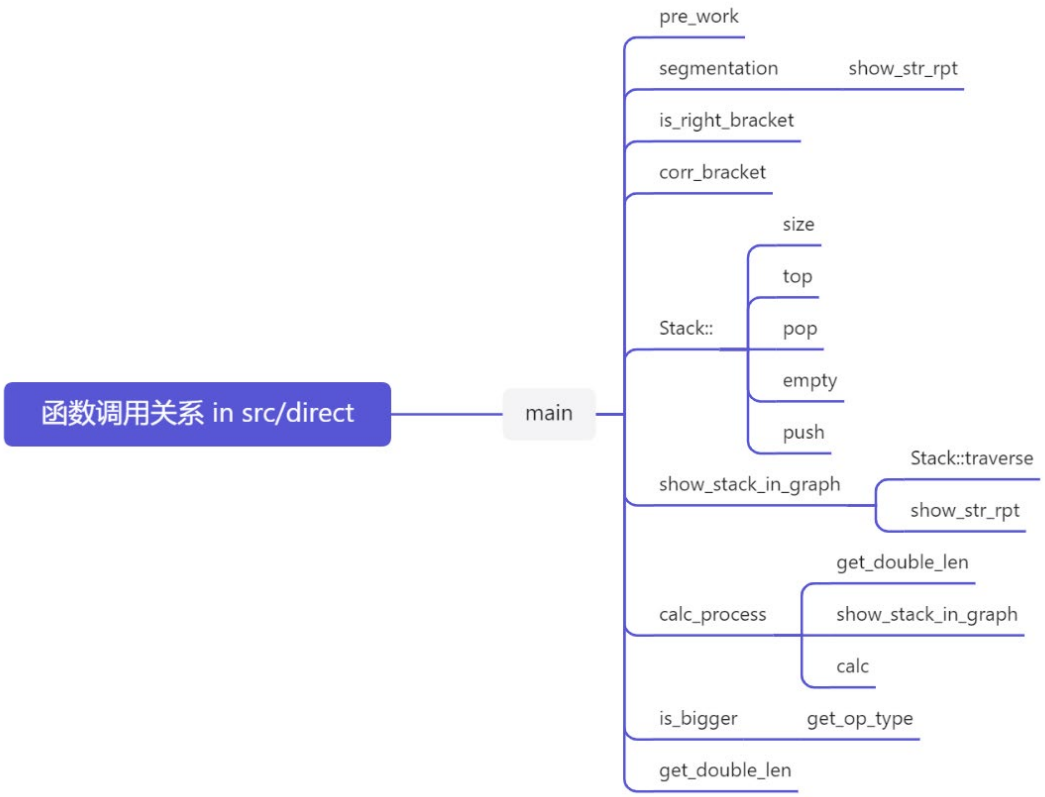
自定义栈　Stack 类模版

```
// 顺序表实现栈
template <typename Elem>
class Stack
{
private:
    SeqList<Elem>* arr = nullptr;

public:
    Stack();
    ~Stack();

    //-------------------------------------
    //自身独有的方法
    //-------------------------------------

    // 入栈
    void push(const Elem& obj);

    // 出栈
    void pop();

    // 栈顶元素
    const Elem& top() const;

    // 是否为空
    bool empty() const;

    // 栈大小
    int size() const;
```

```
30.
31.        // 遍历元素输出，以 separator 为分隔符
32.        void traverse(char separator) const;
33.    };
```







2．物理设计（即存储结构设计）

调用 STL 模版类 vector 作为变长数组

线性表及其派生类设计

（对 MyList 的简单介绍可见./src/header/MyList/README.md）

```
└--> AbstractLinearList (Abstract) // 抽象线性表类
        |
        ├--> SeqList // 顺序表类
        |
        └--> AbstractLinkList (Abstract) // 抽象链表类
                |
                ├--> DynamicLinkList // 动态链表类
                |
                └--> StaticLinkList // 静态链表类

---> DynamicNode // 动态单向节点类
---> StaticNode  // 静态单向节点类
---> Stack       // 栈类（顺序表实现）
---> Queue       // 队列类（用带头结点的单向链表实现）
```

Stack 类模板设计如下

```
└--> Stack<Elem>
        |
        ├-- [private:]
        ├-- SeqList<Elem>* arr
        |
        ├-- [public:]
        ├-- push      // 入栈
        ├-- pop       // 出栈
        ├-- top       // 栈顶元素
        ├-- empty     // 是否为空
        ├-- size      // 栈大小
        └-- traverse  // 遍历元素输出，以separator为分隔符
```

顺序表类模板设计如下

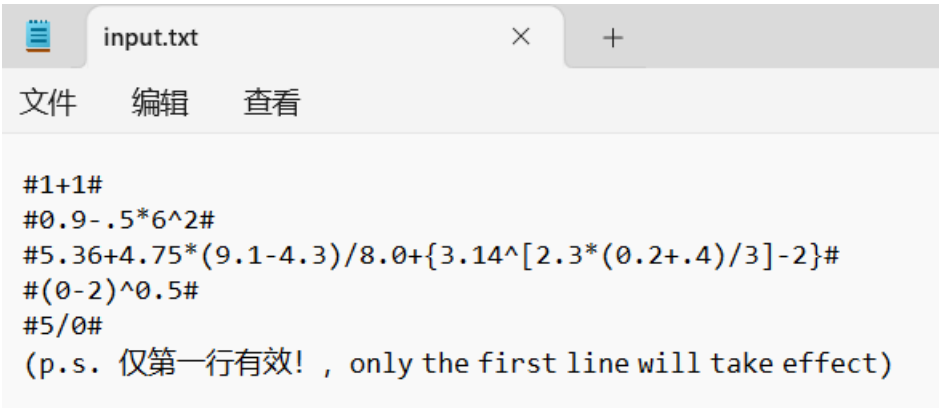
```
├--> SeqList<Elem> : AbstractLinearList<Elem, int>
|        |
|        ├-- [private:]
|        ├-- int used_len     // 已经使用的长度
|        ├-- int malloc_len   // 申请内存的长度
|        ├-- Elem* head_ptr   // 申请顺序表数组地址的指针
|        |
|        ├-- [public:]
|        ├-- ...(11 overwrites)
|        ├-- extend       // 将数组增长len与现有长度一半的最大值的长度
|        ├-- try_extend   // 尝试向数组增加len个元素，如果超出现有长度，调用extend
|        ├-- reverse      // 顺序表就地倒置
|        ├-- cylic_move   // 顺序表循环移动k位，向左为正
|        └-- merge        // 对排序好的两个顺序表进行合并
```

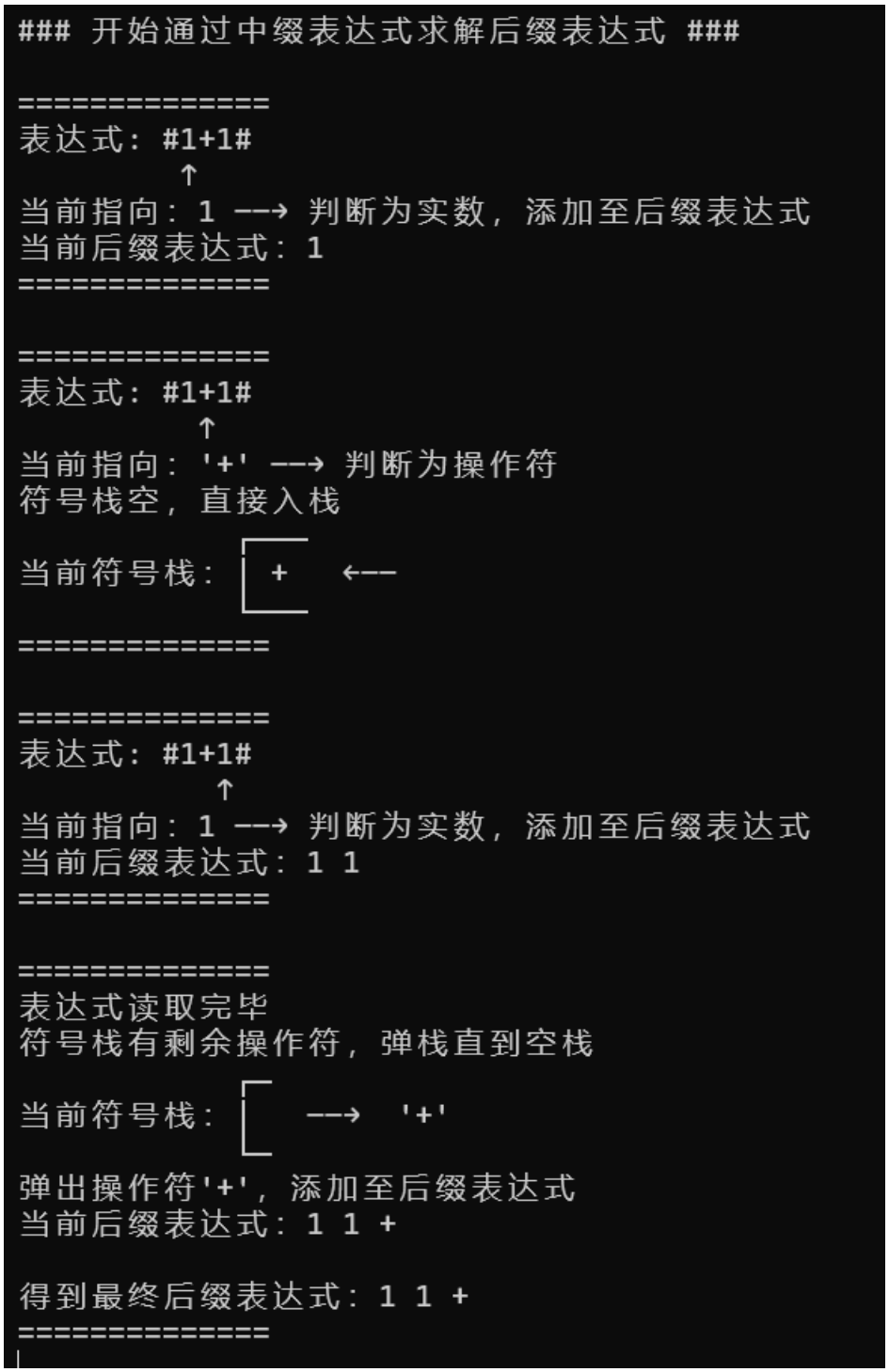
具体内部实现见源代码

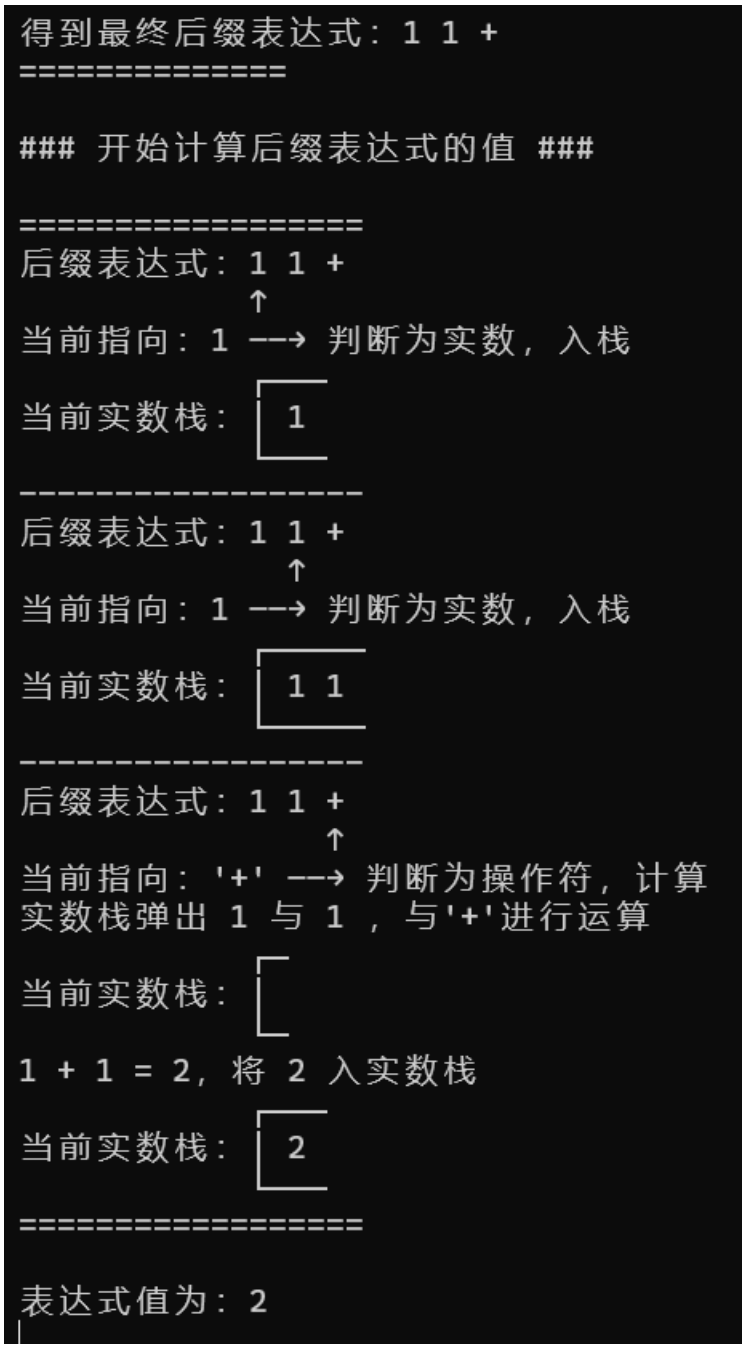
四、测试结果（包括测试数据、结果数据及结果的简单分析和结论，可以用截图得形式贴入此报告）

（以下对 indirect 程序进行测试，direct 程序同理）

test-1：简单表达式的计算




开始通过中缀表达式求解后缀表达式
==========
表达式: #1+1#
↑
当前指向: 1 --> 判断为实数, 添加至后缀表达式
当前后缀表达式: 1
==========
==========
表达式: #1+1#
↑
当前指向: '+' --> 判断为操作符
符号栈空, 直接入栈
当前符号栈: + <--
==========
==========
表达式: #1+1#
↑
当前指向: 1 --> 判断为实数, 添加至后缀表达式
当前后缀表达式: 1 1
==========
==========
表达式读取完毕
符号栈有剩余操作符, 弹栈直到空栈
当前符号栈: --> '+'
弹出操作符'+', 添加至后缀表达式
当前后缀表达式: 1 1 +
得到最终后缀表达式: 1 1 +
==========



结论：正确执行

test-2：复杂表达式的计算

input.txt

文件 编辑 查看

```
#5.36+4.75*(9.1-4.3)/8.0+{3.14^[2.3*(0.2+.4)/3]-2}#
#1+1#
#0.9-.5*6^2#
#(0-2)^0.5#
#5/0#
(p.s. 仅第一行有效! , only the first line will take effect)
```

```
### 开始通过中缀表达式求解后缀表达式 ###

================================================================
表达式: #5.36+4.75*(9.1-4.3)/8.0+{3.14^[2.3*(0.2+.4)/3]-2}#
          ↑
当前指向: 5.36 --> 判断为实数, 添加至后缀表达式
当前后缀表达式: 5.36
================================================================
```

```
============================================================
表达式读取完毕
符号栈有剩余操作符, 弹栈直到空栈

当前符号栈: [   -->  '+'

弹出操作符'+', 添加至后缀表达式
当前后缀表达式: 5.36 4.75 9.1 4.3 - * 8 / + 3.14 2.3 0.2 0.4 + * 3 / ^ 2 - +

得到最终后缀表达式: 5.36 4.75 9.1 4.3 - * 8 / + 3.14 2.3 0.2 0.4 + * 3 / ^ 2 - +
============================================================
```

```
--------------------------------------------------------------------------------
后缀表达式: 5.36 4.75 9.1 4.3 - * 8 / + 3.14 2.3 0.2 0.4 + * 3 / ^ 2 - +
                                                                  ↑
当前指向: '+' --> 判断为操作符, 计算
实数栈弹出 -0.30727 与 8.21 , 与'+'进行运算

当前实数栈: [

8.21 + -0.30727 = 7.90273, 将 7.90273 入实数栈

当前实数栈: [ 7.90273

================================================================================

表达式值为: 7.90273
```

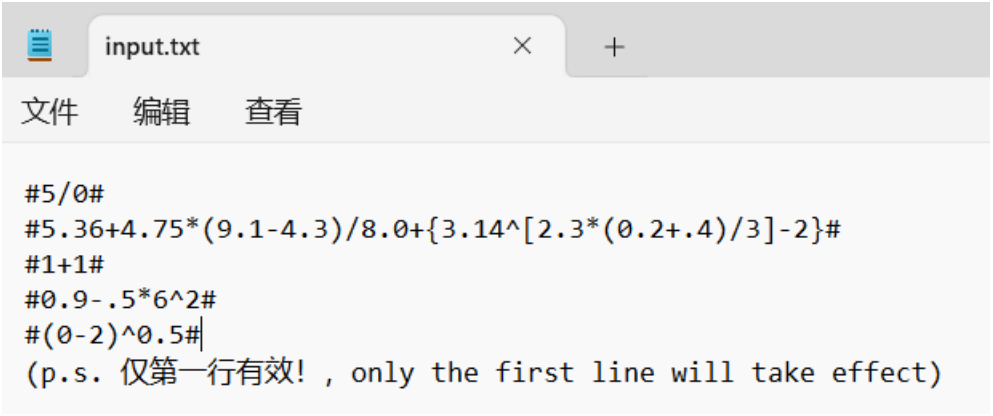
$$5.36+4.75\frac{9.1-4.3}{8}+3.14^{2.3^{\frac{0.2+0.4}{3}}}-2 \quad =7.902729793052325$$

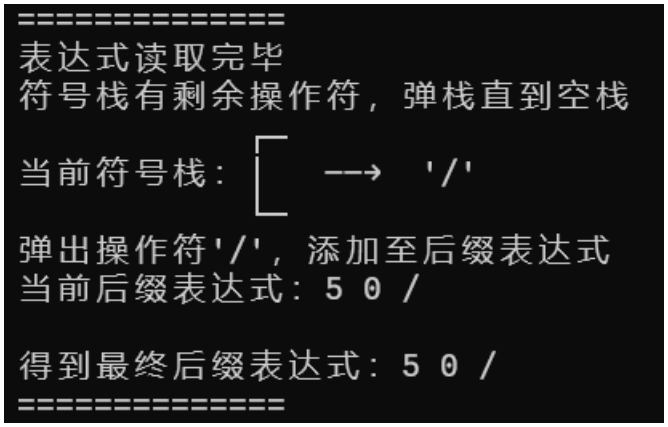
求值

$$\frac{50^{0.54}\times157^{0.46}}{50}+6.21\approx7.902729793$$

结论：计算正确

test-3：除零错误

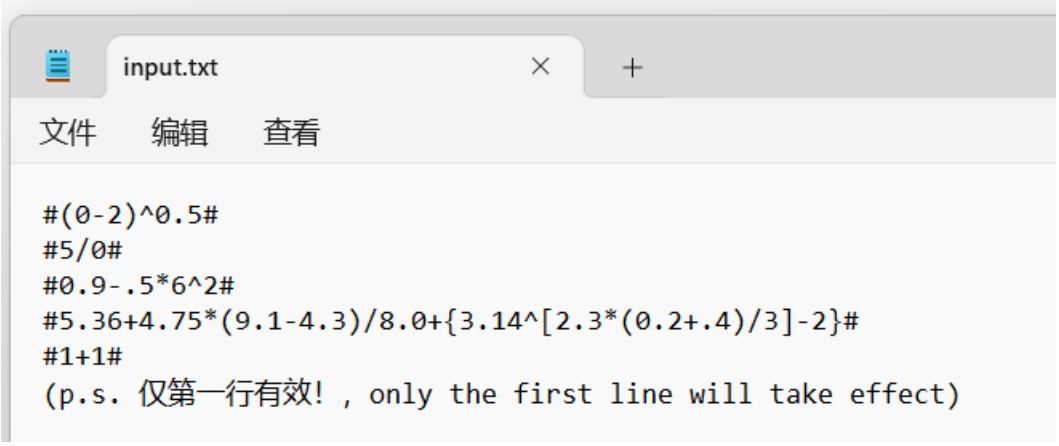


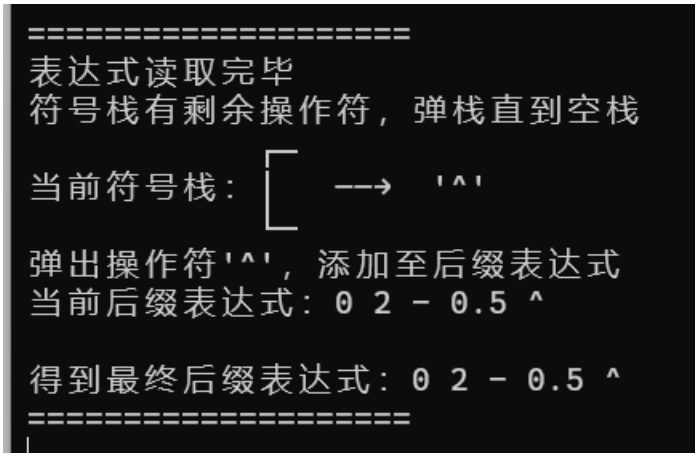


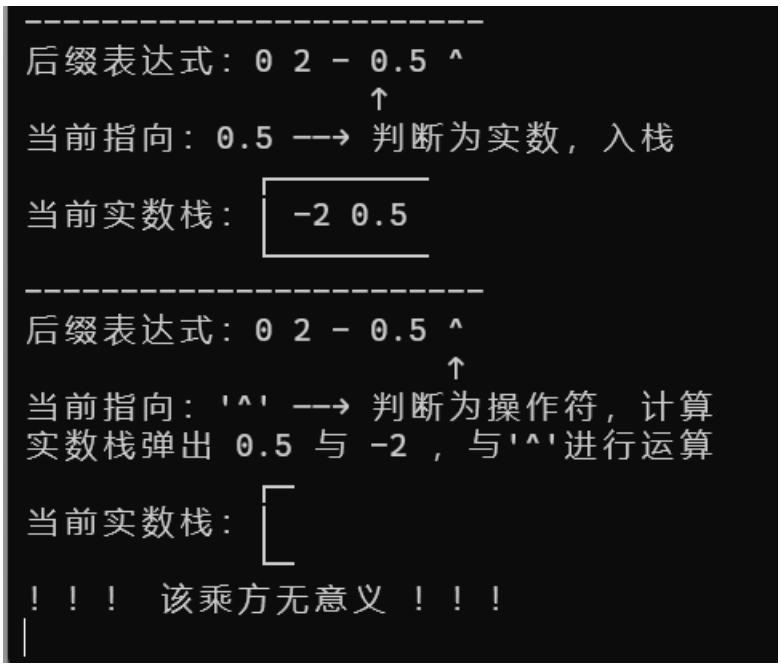
```
### 开始计算后缀表达式的值 ###

==================
后缀表达式: 5 0 /
            ↑
当前指向: 5 --> 判断为实数, 入栈

当前实数栈: ┌──┐
            │ 5│
            └──┘
------------------
后缀表达式: 5 0 /
              ↑
当前指向: 0 --> 判断为实数, 入栈

当前实数栈: ┌────┐
            │ 5 0│
            └────┘
------------------
后缀表达式: 5 0 /
                ↑
当前指向: '/' --> 判断为操作符, 计算
实数栈弹出 0 与 5 , 与'/'进行运算

当前实数栈: ┌┐
            └┘
! ! !  除数不能为零 ! ! !
```

结论：行为符合预期

test-4：求幂错误







结论：行为符合预期

五、经验体会与不足

体会：熟练掌握了表达式求值的过程，体会到栈这种数据结构的重要性

不足：没有以图形界面展现出求解过程

六、附录：源代码（带注释）

./Project/src/indirect/main.cpp

```
1.    using namespace std;
2.
3.    #include "../header/util.h"
4.
```

```
5.    int priority_table[8][8] = {
6.        {0, 0, 0, 0, 0, 1, 1, 1},
7.        {0, 0, 0, 0, 0, 1, 1, 1},
8.        {1, 1, 0, 0, 0, 1, 1, 1},
9.        {1, 1, 0, 0, 0, 1, 1, 1},
10.       {1, 1, 1, 1, 1, 1, 1, 1},
11.       {1, 1, 1, 1, 1, 1, 1, 1},
12.       {1, 1, 1, 1, 1, 1, 1, 1},
13.       {1, 1, 1, 1, 1, 1, 1, 1}
14.   };
15.
16.   int main(int argc, const char *argv[])
17.   {
18.       ifstream input("input.txt", ios::binary);
19.       string expr;
20.       int len;
21.
22.       // get the whole infix expression, and the length of it
23.       pre_work(input, expr, len);
24.
25.       // stack for operators
26.       Stack<char> st_op;
27.
28.       // vector for suffix expression
29.       vector<Term> suffix_expr;
30.
31.       // discard the character '#'
32.       input.get();
33.
34.       // start progress 1 : calc out the suffix expression
35.       cout << "### 开始通过中缀表达式求解后缀表达式 ###\n" << endl;
36.
37.       char ch;
38.       while (ch = input.peek(), ch != '#')
39.       // peek the initial character of the term
40.       {
41.           segmentation('=', len + 9);
42.
43.           // show the reading expression
44.           cout << "表达式: " << expr << endl;
45.
46.           // show the arrow "↑" pointing to the term which is being
          reading
```

```
47.           cout << string((int)input.tellg() + 8, ' ') << "↑" << end
          l;
48.
49.           if (isdigit(ch) || ch == '.')
50.           // if the term is a double
51.           // i.e. the term with the beginning character of "0123456
          789."
52.           // should be push back into the suffix expr
53.           {
54.               double num;
55.               // read the double
56.               input >> num;
57.
58.               // show progress
59.               cout << "当前指向：" << num << " ⟶ 判断为实数，添加至后
          缀表达式" << endl;
60.
61.               // push back the term into the suffix expression
62.               suffix_expr.emplace_back(Term(NUM, to_string(num)));
63.
64.               // show current suffix expression
65.               show_suffix_expr("当前后缀表达式：", suffix_expr);
66.           }
67.           else if (is_right_bracket(ch))
68.           // if the term is a right bracket
69.           // i.e. be of ")]}"
70.           // the operator stack should pop until meet the correspon
          ding bracket
71.           // i.e. be of "([{"
72.           {
73.               // discard the character
74.               input.get();
75.
76.               // show progress
77.               cout << "当前指向：\'" << ch << "\' ⟶ 判断为右括号，应
          弹出符号栈元素，直到遇到左括号
          \'" << corr_bracket(ch) << '\'' << endl;
78.
79.               // show current operator stack
80.               show_stack_in_graph(st_op, ' ', 0, 0, st_op.size() *
          2 + 1, "当前符号栈");
81.               getch();
82.
83.               char op;
```

```
84.             while (op = st_op.top(), st_op.pop(), op != corr_brac
        ket(ch))
85.             // the operator stack should pop until meet the corre
        sponding left bracket
86.             {
87.                 // show progress
88.                 cout << "符号栈中弹出符号：\'" << op << "\'，添加至后
        缀表达式" << endl;
89.
90.                 // push back the popped operator into the suffix
        expr
91.                 suffix_expr.emplace_back(Term(OPERATOR, op));
92.
93.                 // show current suffix expression
94.                 show_suffix_expr("当前后缀表达式：", suffix_expr);
95.
96.                 // show current operator stack
97.                 show_stack_in_graph(st_op, ' ', 2, op, st_op.size
        () * 2 + 1, "当前符号栈");
98.                 getch();
99.
100.                segmentation('-', len + 9);
101.            }
102.            // show the corresponding left bracket, and the progr
        ess
103.            cout << "符号栈中弹出符号：\'" << op << "\'，结束弹栈
        " << endl;
104.
105.            // show current operator stack
106.            show_stack_in_graph(st_op, ' ', 2, op, st_op.size() *
         2 + 1, "当前符号栈");
107.
108.            // show current suffix expression
109.            show_suffix_expr("当前后缀表达式：", suffix_expr);
110.        }
111.        else
112.        // if the term is a operator
113.        // i.e. be of "+-*/^([{"
114.        // push into the operator stack, obeying the rule "the up
        per op's priority should be greater than its lower op's"
115.        {
116.            // discard the character
117.            input.get();
118.
```

```
119.                // show progress
120.                cout << "当前指向：\'" << ch << "\' ⟶ 判断为操作符
    " << endl;
121.                while (1)
122.                {
123.                    if (st_op.empty() || is_bigger(ch, st_op.top()))
124.                    // op-stack is empty, or the reading character is
     "bigger" than the top one
125.                    // push in direct
126.                    {
127.                        // show progress
128.                        if (st_op.empty())
129.                            cout << "符号栈空，直接入栈" << endl;
130.                        else
131.                            cout << "当前操作符\'" << ch << "\'的优先级
    大于符号栈顶的操作符：\'" << st_op.top() << "\'，将
    \'" << ch << "\'入栈" << endl;
132.
133.                        st_op.push(ch);
134.
135.                        // show current operator stack
136.                        show_stack_in_graph(st_op, ' ', 1, 0, st_op.s
    ize() * 2 + 1, "当前符号栈");
137.
138.                        // over
139.                        break;
140.                    }
141.                    else
142.                    // the reading character is "not bigger" than the
     top one
143.                    // pop the op-stack until the the reading charact
    er is "bigger" than the top one
144.                    {
145.                        // get op-stack top
146.                        char temp = st_op.top();
147.
148.                        // show progress
149.                        cout << "当前操作符\'" << ch << "\'的优先级不大
    于符号栈顶的操作符：\'" << temp << "\'，符号栈弹栈" << endl;
150.
151.                        // push back the popped operator into the suf
    fix expr
152.                        suffix_expr.emplace_back(Term(OPERATOR, temp)
    );
```

```
153.
154.                     // show current suffix expression
155.                     show_suffix_expr("当前后缀表达式:
          ", suffix_expr);
156.
157.                     // op-stack pop
158.                     st_op.pop();
159.
160.                     // show current operator stack
161.                     show_stack_in_graph(st_op, ' ', 2, temp, st_o
          p.size() * 2 + 1, "当前符号栈");
162.                     getch();
163.
164.                     segmentation('-', len + 9);
165.
166.                     // pop the op-stack until the the reading cha
          racter is "bigger" than the top one
167.                 }
168.             }
169.         }
170.         segmentation('=', len + 9, '\n');
171.         getch();
172.     }
173.     input.close();
174.     segmentation('=', len + 9);
175.
176.     // show progress
177.     cout << "表达式读取完毕" << endl;
178.
179.     if (!st_op.empty())
180.     // if there are some operators left in the op-stack
181.     {
182.         // show progress
183.         cout << "符号栈有剩余操作符，弹栈直到空栈" << endl;
184.
185.         while (!st_op.empty())
186.         // while there are some operators left in the op-stack
187.         {
188.             // get top and pop
189.             char temp = st_op.top();
190.             st_op.pop();
191.
192.             // show current operator stack
```

```
193.                 show_stack_in_graph(st_op, ' ', 2, temp, st_op.size()
             * 2 + 1, "当前符号栈");
194.
195.                 // show progress
196.                 cout << "弹出操作符\'" << temp << "\'，添加至后缀表达式
            " << endl;
197.
198.                 // push back the popped operator into the suffix expr
199.                 suffix_expr.emplace_back(Term(OPERATOR, temp));
200.
201.                 // show current suffix expression
202.                 show_suffix_expr("当前后缀表达式：", suffix_expr);
203.                 getch();
204.
205.                 if (st_op.size() != 0)
206.                     segmentation('-', len + 9);
207.             }
208.         }
209.         cout << endl;
210.
211.         // show final suffix expression
212.         show_suffix_expr("得到最终后缀表达式：", suffix_expr);
213.
214.         segmentation('=', len + 9);
215.         getch();
216.
217.         //============================================================
              ==================
218.
219.         // start progress 2 : calc the value of the expression
220.         cout << "\n### 开始计算后缀表达式的值 ###\n" << endl;
221.
222.         // stack for operation nums
223.         Stack<double> st_num;
224.
225.         // calc the title len (just for aesthetics)
226.         int title_len = get_suffix_expr_len(suffix_expr);
227.
228.         // show title, 12 is the len of "后缀表达式："
229.         segmentation('=', title_len + 12);
230.
231.         int arrow_index = 12;
232.
233.         // the len of "—" to be illustrated
```

```
234.        int stack_boundary_len = 1;
235.
236.        for (int i = 0; i < suffix_expr.size(); ++i)
237.        {
238.            // show suffix expression
239.            show_suffix_expr("后缀表达式: ", suffix_expr);
240.
241.            // show arrow pointing to the term being read
242.            cout << string(arrow_index, ' ') << "↑" << endl;
243.
244.            Term& now_term = suffix_expr.at(i);
245.            if (now_term.type == NUM)
246.            // the term is a double (operand)
247.            {
248.                // get the double value
249.                double temp = stod(now_term.str);
250.
251.                // show progress
252.                cout << "当前指向: " << temp << " ⟶ 判断为实数，入栈
            " << endl;
253.
254.                // push into the operand stack
255.                st_num.push(temp);
256.
257.                // increase the stack boundary length, the extra 1 is
             for the separator ' '
258.                stack_boundary_len += (get_double_len(temp) + 1);
259.
260.                // show current operand stack
261.                show_stack_in_graph(st_num, ' ', 0, 0, stack_boundary
          _len, "当前实数栈");
262.
263.                // increase the arrow index, let the arrow point to t
          he next term, the extra 1 is for the separator ' '(space)
264.                arrow_index += (get_double_len(temp) + 1);
265.            }
266.            else
267.            // the term is a operator
268.            {
269.                // show progress
270.                cout << "当前指向:\'" << now_term.str << "\' ⟶ 判断为
          操作符，计算" << endl;
271.
```

```
272.                 calc_process(st_num, stack_boundary_len, now_term.str
            [0]);
273.
274.                 // increase the arrow index, let the arrow point to t
            he next term
275.                 // for 2, one is for the operator, the other one is f
            or separator ' '(space)
276.                 arrow_index += 2;
277.             }
278.
279.             if (i != suffix_expr.size() - 1)
280.             // if it is not the last term
281.                 // print the segmentation
282.                 segmentation('-', title_len + 12);
283.
284.             getch();
285.         }
286.         segmentation('=', title_len + 12, '\n');
287.
288.         // show final result
289.         cout << "表达式值为: " << st_num.top() << endl;
290.         getch();
291.
292.         return 0;
293.     }
```

./Project/src/direct/main.cpp

```
1.      using namespace std;
2.
3.      #include "../header/util.h"
4.
5.      int priority_table[8][8] = {
6.          {0, 0, 0, 0, 0, 1, 1, 1},
7.          {0, 0, 0, 0, 0, 1, 1, 1},
8.          {1, 1, 0, 0, 0, 1, 1, 1},
9.          {1, 1, 0, 0, 0, 1, 1, 1},
10.         {1, 1, 1, 1, 1, 1, 1, 1},
11.         {1, 1, 1, 1, 1, 1, 1, 1},
12.         {1, 1, 1, 1, 1, 1, 1, 1},
13.         {1, 1, 1, 1, 1, 1, 1, 1}
14.     };
15.
```

```
16.  int main(int argc, const char *argv[])
17.  {
18.      ifstream input("input.txt", ios::binary);
19.      string expr;
20.      int len;
21.
22.      // get the whole infix expression, and the length of it
23.      pre_work(input, expr, len);
24.
25.      // stack for operators
26.      Stack<char> st_op;
27.
28.      // stack for nums
29.      Stack<double> st_num;
30.
31.      // discard the character '#'
32.      input.get();
33.
34.      // start progress : calc the value of the infix expression
35.      cout << "### 开始计算中缀表达式的值 ###\n" << endl;
36.
37.      char ch;
38.
39.      // the len of "—" to be illustrated
40.      int stack_boundary_len = 1;
41.
42.      while (ch = input.peek(), ch != '#')
43.      // peek the initial character of the term
44.      {
45.          segmentation('=', len + 9);
46.
47.          // show the reading expression
48.          cout << "表达式: " << expr << endl;
49.
50.          // show the arrow "↑" pointing to the term which is being
          reading
51.          cout << string((int)input.tellg() + 8, ' ') << "↑" << end
          l;
52.
53.          if (isdigit(ch) || ch == '.')
54.          // if the term is a double
55.          // i.e. the term with the beginning charactor of "0123456
          789."
56.          // should be push into the op-num-stack
```

```
57.           {
58.               double num;
59.               // read the double
60.               input >> num;
61.
62.               // show progress
63.               cout << "当前指向：" << num << " ⟶ 判断为实数，入实数栈
          " << endl;
64.
65.               // push the num into the op-num-stack
66.               st_num.push(num);
67.
68.               // increase the op-num-stack boundary length, the ext
          ra 1 is for the separator ' '
69.               stack_boundary_len += (get_double_len(num) + 1);
70.
71.               // show current op-num-stack
72.               show_stack_in_graph(st_num, ' ', 0, 0, stack_boundary
          _len, "当前实数栈");
73.           }
74.           else if (is_right_bracket(ch))
75.           // if the term is a right bracket
76.           // i.e. be of ")]}"
77.           // the operator stack should pop until meet the correspon
          ding bracket
78.           // i.e. be of "([{"
79.           {
80.               // discard the character
81.               input.get();
82.
83.               // show progress
84.               cout << "当前指向：\'" << ch << "\' ⟶ 判断为右括号，应
          弹出符号栈元素，直到遇到左括号
          \'" << corr_bracket(ch) << '\'' << endl;
85.
86.               // show current operator stack
87.               show_stack_in_graph(st_op, ' ', 0, 0, st_op.size() *
          2 + 1, "当前符号栈");
88.               getch();
89.
90.               segmentation('-', len + 9);
91.
92.               char op;
```

```
93.                 while (op = st_op.top(), st_op.pop(), op != corr_brac
            ket(ch))
94.                 // the operator stack should pop until meet the corre
            sponding left bracket
95.                 {
96.                     // show progress
97.                     cout << "符号栈中弹出符号：\'" << op << "\'，进行计算
            " << endl;
98.
99.                     // show current op-stack
100.                    show_stack_in_graph(st_op, ' ', 2, op, st_op.size
            () * 2 + 1, "当前符号栈");
101.
102.                    calc_process(st_num, stack_boundary_len, op);
103.                    getch();
104.
105.                    segmentation('-', len + 9);
106.                }
107.                // show the corresponding left bracket, and the progr
            ess
108.                cout << "符号栈中弹出符号：\'" << op << "\'，结束弹栈
            " << endl;
109.
110.                // show current operator stack
111.                show_stack_in_graph(st_op, ' ', 2, op, st_op.size() *
             2 + 1, "当前符号栈");
112.                // show current op-num-stack
113.                show_stack_in_graph(st_num, ' ', 0, 0, stack_boundary
            _len, "当前实数栈");
114.            }
115.            else
116.            // if the term is a operator
117.            // i.e. be of "+-*/^([{"
118.            // push into the operator stack, obeying the rule "the up
            per op's priority should be greater than its lower op's"
119.            {
120.                // discard the character
121.                input.get();
122.
123.                // show progress
124.                cout << "当前指向：\'" << ch << "\' ⟶ 判断为操作符
            " << endl;
125.                while (1)
126.                {
```

```
127.                 if (st_op.empty() || is_bigger(ch, st_op.top()))
128.                 // op-stack is empty, or the reading character is "bigger" than the top one
129.                 // push in direct
130.                 {
131.                     // show progress
132.                     if (st_op.empty())
133.                         cout << "符号栈空，直接入栈" << endl;
134.                     else
135.                         cout << "当前操作符\'" << ch << "\'的优先级大于符号栈顶的操作符：\'" << st_op.top() << "\'，将\'" << ch << "\'入栈" << endl;
136.
137.                     st_op.push(ch);
138.
139.                     // show current operator stack
140.                     show_stack_in_graph(st_op, ' ', 1, 0, st_op.size() * 2 + 1, "当前符号栈");
141.
142.                     // over
143.                     break;
144.                 }
145.                 else
146.                 // the reading character is "not bigger" than the top one
147.                 // pop the op-stack until the the reading character is "bigger" than the top one
148.                 {
149.                     // get op-stack top
150.                     char temp = st_op.top();
151.
152.                     // show progress
153.                     cout << "当前操作符\'" << ch << "\'的优先级不大于符号栈顶的操作符：\'" << temp << "\'，符号栈弹栈，进行计算" << endl;
154.
155.                     // op-stack pop
156.                     st_op.pop();
157.
158.                     // show current operator stack
159.                     show_stack_in_graph(st_op, ' ', 2, temp, st_op.size() * 2 + 1, "当前符号栈");
160.
```

```
                    calc_process(st_num, stack_boundary_len, temp);
                    getch();

                    segmentation('-', len + 9);
                    // pop the op-stack until the the reading character is "bigger" than the top one
                }
            }
        }
        segmentation('=', len + 9, '\n');
        getch();
    }
    input.close();

    segmentation('=', len + 9);

    if (!st_op.empty())
    // if there are some operators left in the op-stack
    {
        // show progress
        cout << "符号栈有剩余操作符，弹栈直到空栈" << endl;

        while (!st_op.empty())
        // while there are some operators left in the op-stack
        {
            // get top and pop
            char temp = st_op.top();
            st_op.pop();

            // show progress
            cout << "弹出操作符\'" << temp << "\'，进行计算" << endl;

            // show current operator stack
            show_stack_in_graph(st_op, ' ', 2, temp, st_op.size() * 2 + 1, "当前符号栈");

            calc_process(st_num, stack_boundary_len, temp);
            getch();

            // segmentation
            if (st_op.size() != 0)
                segmentation('-', len + 9);
```

```
            }
        }
        segmentation('=', len + 9);

        // show result
        cout << "\n 表达式读取完毕，结果为" << st_num.top() << endl;
        getch();

        return 0;
    }
```

./Project/src/header/util.h

```
#ifndef _UTIL_H_INCLUDED_
#define _UTIL_H_INCLUDED_

#include <iostream>
#include "../header/MyList/Stack.hpp"
#include <vector>
#include <string>
#include <fstream>
#include <cmath>
#include <conio.h>

extern int priority_table[8][8];

enum TermType
{
    NUM = 0,
    OPERATOR
};

class Term
{
public:
    TermType type;
    std::string str;
    Term(TermType type, std::string str): type(type), str(str) {}
    Term(TermType type, char ch) : type(type), str(1, ch) {}
};

void pre_work(std::ifstream& input, std::string& expr, int& len);

```

```
31.   void show_suffix_expr(std::string memo, std::vector<Term> &v);
32.   template <typename T>
33.   void show_stack_in_graph(Stack<T> &st, char separator, int type,
            char elem, int boundary_len, const std::string& memo);
34.
35.   void show_str_rpt(std::string str, int times);
36.   void segmentation(std::string rpt_str, int rpt_cnt, char suffix =
            '\0');
37.   void segmentation(char rpt_char, int rpt_cnt, char suffix = '\0')
            ;
38.
39.   int is_bigger(char op_1, char op_2);
40.   int get_op_type(char op);
41.   int is_right_bracket(char bracket);
42.   char corr_bracket(char right_bracket);
43.   double calc(double a, double b, const std::string& oper);
44.   int get_double_len(double num);
45.   int get_suffix_expr_len(std::vector<Term>& suffix_expr);
46.
47.   void calc_process(Stack<double>& st_num, int& stack_boundary_len,
            char op);
48.
49.   // the memo should be 6 Chinese charactor for formating
50.   template <typename T>
51.   void show_stack_in_graph(Stack<T> &st, char separator, int type,
            char elem, int boundary_len, const std::string& memo)
52.   {
53.       std::cout << "            ┌"; show_str_rpt("─", boundary_len)
            , std::cout << std::endl;
54.       std::cout << memo << ": │ "; st.traverse(' ');
55.
56.       switch (type)
57.       {
58.       case 0:
59.           std::cout << std::endl;
60.           break;
61.       case 1:
62.           std::cout << "  ←──" << std::endl;
63.           break;
64.       case 2:
65.           std::cout << "  ──→  \'" << elem << "\'" << std::endl;
66.           break;
67.       }
```

```
68.         std::cout << "            └"; show_str_rpt("─", boundary_len)
              , std::cout << std::endl;
69.     }
70.
71.     inline void show_str_rpt(std::string str, int times)
72.     {
73.         for (int i = 0; i < times; i++)
74.             std::cout << str;
75.     }
76.
77.     inline void segmentation(std::string rpt_str, int rpt_cnt, char s
            uffix)
78.     {
79.         show_str_rpt(rpt_str, rpt_cnt);
80.         std::cout << suffix << std::endl;
81.     }
82.
83.     inline void segmentation(char rpt_char, int rpt_cnt, char suffix)
84.     {
85.         std::cout << std::string(rpt_cnt, rpt_char) << suffix << std:
              :endl;
86.     }
87.
88.     inline int is_bigger(char op_1, char op_2)
89.     {
90.         int i = get_op_type(op_1), j = get_op_type(op_2);
91.         return priority_table[i][j];
92.     }
93.
94.     inline int get_op_type(char op)
95.     {
96.         std::string str = "+-*/^([{";
97.         return str.find(op);
98.     }
99.
100.    inline int is_right_bracket(char bracket)
101.    {
102.        return bracket == ')' || bracket == ']' || bracket == '}';
103.    }
104.
105.    inline char corr_bracket(char right_bracket)
106.    {
107.        switch (right_bracket)
108.        {
```

```
        case ')':
            return '(';
        case ']':
            return '[';
        case '}':
            return '{';
        default:
            return '\0';
        }
    }

    #endif
```

./Project/src/header/util.cpp

```
    #include "util.h"

    void pre_work(std::ifstream& input, std::string& expr, int& len)
    {
        // set terminal to code UTF-8
        system("chcp 65001");

        system("cls");

        // check the validity of the file
        if (input.peek() != '#')
        {
            std::cout << "Input file format error!" << std::endl;
            getch();
            exit(-1);
        }

        // read whole expression, for show purpose
        input >> expr;
        len = expr.length();

        // set the file reading pointer to the beginning of the file
        input.seekg(0);
    }

    // print terms in suffix_expr
    void show_suffix_expr(std::string memo, std::vector<Term> &v)
    {
```

```
29.       std::cout << memo;
30.       for (auto term: v)
31.       {
32.           if (term.type == 0)
33.               std::cout << stod(term.str) << ' ';
34.           else
35.               std::cout << term.str << ' ';
36.       }
37.       std::cout << std::endl;
38.   }
39.
40.   double calc(double a, double b, const std::string& oper)
41.   {
42.       if (oper == "+")
43.           return a + b;
44.       else if (oper == "-")
45.           return a - b;
46.       else if (oper == "*")
47.           return a * b;
48.       else if (oper == "/")
49.       {
50.           if (std::isinf(a / b))
51.               throw "! ! !  除数不能为零 ! ! ! ";
52.           else
53.               return a / b;
54.       }
55.       else if (oper == "^")
56.       {
57.           if (!std::isnormal(pow(a, b)))
58.               throw "! ! !  该乘方无意义 ! ! ! ";
59.           else
60.               return pow(a, b);
61.       }
62.       else
63.           return 0;
64.   }
65.
66.   int get_double_len(double num)
67.   {
68.       num = (int)(num * 100000) / 100000.0;
69.
70.       int len = 0;
71.
72.       if (num < 0)
```

```
    {
        ++len;
        num = -num;
    }
    int integer = round(num);

    do
    {
        ++len;
        integer /= 10;
    }
    while (integer > 0);

    double decimals = num - round(num);
    decimals = fabs(decimals);
    if (decimals > 1e-6)
    {
        ++len;
        do
        {
            ++len;
            decimals *= 10;
            decimals -= round(decimals);
            decimals = fabs(decimals);
        }
        while (decimals > 1e-4);
    }

    return len;
}

int get_suffix_expr_len(std::vector<Term>& suffix_expr)
{
    int len = 0;
    for (int i = 0; i < suffix_expr.size(); i++)
    {
        Term &temp = suffix_expr.at(i);
        if (temp.type == NUM)
            len += get_double_len(stod(temp.str));
        else // temp.type == OPERATOR
            len += 1;
    }
    len += suffix_expr.size();
    return len;
```

```
117. }
118.
119. void calc_process(Stack<double>& st_num, int& stack_boundary_len,
         char op)
120. {
121.     // get the two op-num out, decrease the stack boundary length
       , the extra 1 is for the separator ' '
122.     double num2 = st_num.top();
123.     st_num.pop();
124.     stack_boundary_len -= (get_double_len(num2) + 1);
125.     double num1 = st_num.top();
126.     st_num.pop();
127.     stack_boundary_len -= (get_double_len(num1) + 1);
128.
129.     // show progress
130.     std::cout << "实数栈弹出 " << num2 << " 与 " << num1 << " , 与
       \'" << op << "\'进行运算" << std::endl;
131.
132.     // show current operand stack
133.     show_stack_in_graph(st_num, ' ', 0, 0, stack_boundary_len, "
       当前实数栈");
134.
135.     // calc the result
136.     double res;
137.     try
138.     {
139.         res = calc(num1, num2, std::string(1, op));
140.     }
141.     catch (const char* str)
142.     // math error
143.     {
144.         std::cout << str << std::endl;
145.         getch();
146.         exit(-1);
147.     }
148.
149.     // show progress
150.     std::cout << num1 << ' ' << op << ' ' << num2 << " = " << res
       << ", 将 " << res << " 入实数栈" << std::endl;
151.
152.     // push the result into the operand stack, increase the stack
       boundary length
153.     st_num.push(res);
154.     stack_boundary_len += (get_double_len(res) + 1);
```

```
155.
156.     // show current operand stack
157.     show_stack_in_graph(st_num, ' ', 0, 0, stack_boundary_len, "
           当前实数栈");
158. }
```

./Project/src/header/MyList/Stack.hpp

```
1.  #ifndef STACK_HPP_INCLUDED
2.  #define STACK_HPP_INCLUDED
3.
4.  #include "SeqList.hpp"
5.
6.  // 顺序表实现栈
7.  template <typename Elem>
8.  class Stack
9.  {
10. private:
11.     SeqList<Elem>* arr = nullptr;
12.
13. public:
14.     Stack();
15.     ~Stack();
16.
17.     //-------------------------------------
18.     //自身独有的方法
19.     //-------------------------------------
20.
21.     // 入栈
22.     void push(const Elem& obj);
23.
24.     // 出栈
25.     void pop();
26.
27.     // 栈顶元素
28.     const Elem& top() const;
29.
30.     // 是否为空
31.     bool empty() const;
32.
33.     // 栈大小
34.     int size() const;
35.
```

```
36.      // 遍历元素输出，以 separator 为分隔符
37.      void traverse(char separator) const;
38.  };
39.
40.  #include "Stack.cpp"
41.
42.  #endif
```

./Project/src/header/MyList/Stack.cpp

```
1.   #include "Stack.hpp"
2.
3.   // 构造函数
4.   template <typename Elem>
5.   Stack<Elem>::Stack()
6.   {
7.       arr = new SeqList<Elem>;
8.   }
9.
10.  // 析构函数
11.  template <typename Elem>
12.  Stack<Elem>::~Stack()
13.  {
14.      if (arr)
15.          delete arr;
16.  }
17.
18.  // 入栈
19.  template <typename Elem>
20.  void Stack<Elem>::push(const Elem& obj)
21.  {
22.      arr->push_back(obj);
23.  }
24.
25.  // 出栈
26.  template <typename Elem>
27.  void Stack<Elem>::pop()
28.  {
29.      if (! arr->empty())
30.          arr->remove(arr->prior(arr->end()));
31.  }
32.
33.  // 栈顶元素
```

```
34.  template <typename Elem>
35.  const Elem& Stack<Elem>::top() const
36.  {
37.      // 空栈查看栈顶是未定义行为
38.      return arr->at(arr->prior(arr->end()));
39.  }
40.
41.  // 是否为空
42.  template <typename Elem>
43.  bool Stack<Elem>::empty() const
44.  {
45.      return arr->empty();
46.  }
47.
48.  // 栈大小
49.  template <typename Elem>
50.  int Stack<Elem>::size() const
51.  {
52.      return arr->length();
53.  }
54.
55.  // 遍历元素输出，以 separator 为分隔符
56.  template <typename Elem>
57.  void Stack<Elem>::traverse(char separator) const
58.  {
59.      arr->traverse(separator);
60.  }
```