实验报告（二）

班级：2022217803
姓名：密言
学号：2022210064

2024.3.25

第一部分　括弧匹配问题

1　需求分析

1.1　题目

编写程序，从标准输入得到一个缺失左括号的表达式，输出补全括号后的中序表达式。
例如：输入 1+2)*3-4)*5-6)))，则程序输出 ((1+2)*((3-4)*(5-6)))。

1.2　分析

此题可以使用栈来解决，利用栈 LIFO (Last-In-First-Out) 的原则，将操作数和操作符分别压栈出栈，最后操作数栈中剩下的元素即为完整的表达式。
输入为一个仅缺少左括号的表达式，表达式中的数字可以是任意正整数。若输入表达式可以正确补全，输出为补全左括号之后的完整表达式，否则输出为 Conversion failed. The input expression may be invalid.
下面给出有效输入和无效输入及其输出结果的示例：
input(1):
1+2)*3-4)*5-6)))
output(1):
((1+2)*((3-4)*(5-6)))
input(2): 1+)2
output(2):
Conversion failed. The input expression may be invalid.

2　概要设计

使用栈数据结构，创建两个栈 dataStack（数字栈）和 operStack（符号栈），栈使用结构体类型，分别用来存放数字和运算符。程序从前到后遍历整个字符串，如果没识别到”)”，则数字入数字栈，符号入符号栈。遇到右括号时，将两个操作数和一个符号出栈，加上左右括号拼接成一个

字符串整体，把其当作数字入数字栈。直到数字栈中只剩下一个元素，即为补全左括号之后的表达式。

3 详细设计

3.1 栈的初始化和操作函数

使用结构体定义一个栈，表达式数据由二维数组存储，整型变量存储栈顶位置。

```
typedef struct Stack {
char data[MAX_EXPR_LEN][MAX_EXPR_LEN * 2];
int top;
} Stack;

// 初始化栈
void initStack(Stack* s) {
    s->top = -1;
}

// 入栈操作
void Push(Stack* s, char* data) {
    s->top++;
    strcpy(s->data[s->top], data);
}

// 出栈操作
char* Pop(Stack* s) {
    return s->data[s->top--];
}
```

3.2 补全括号函数

这个函数是本题中的关键函数。首先初始化数字栈和符号栈，然后遍历表达式。遍历过程中无非出现以下四种情况，用 if-else if-else 语句嵌套：
如果遇到数字，就将数字串存在 tempData 数组中，直到遍历到非数字字符，说明连续的数字串识别结束，数字入数字栈。
如果遇到运算符，就将运算符存到符号栈中。
如果遇到右括号，就将数字栈顶前两个数出栈，分别为右操作数和左操作数（这里有特殊情况，如果出现表达式嵌套多层括号的情况，没有“左操作数”，此时应该直接把“右操作数”加上括号入数字栈），符号栈栈顶的一个元素出栈，为运算符。将它们拼接成为一个字符串，加上左右括号，作为整体入数字栈。

如果遇到其他情况，说明表达式有误，直接返回 false。
上述情况中都需要判断数字栈和符号栈中的元素数量是否差 1，即两个数字和一个运算符。如果不满足条件，则说明表达式有误。
遍历结束后，如果数字栈和符号栈中仍有剩余元素，说明结尾没有后括号，此时直接拼接操作数和操作符（不需要加括号，否则会出现运算逻辑错误），然后入数字栈，直到数字栈中只剩下一个元素，即为补全括号后的表达式。
函数代码如下：

```
    // 补全括号函数
bool completeParentheses(char* expression, char* result) {
    int exprLength = strlen(expression);

    if (exprLength >= MAX_EXPR_LEN) {
        return false;
    }

    char tempData[MAX_EXPR_LEN * 2] = {0};
    Stack dataStack, operStack;

    initStack(&dataStack);
    initStack(&operStack);

    int j = 0;

    // 遍历表达式
    for (int i = 0; i < exprLength; i++) {
        char element = expression[i];

        if (element >= '0' && element <= '9') {
            // 获取操作数
            tempData[j++] = element;
            i++;

            while (i < exprLength) {
                char element1 = expression[i];

                if (element1 >= '0' && element1 <= '9') {
                    tempData[j++] = element1;
                    i++;
                } else {
                    i--;
                    break;
```

```
35                  }
36              }
37
38              Push(&dataStack, tempData);
39
40              // 栈顶元素判断
41              if (dataStack.top - operStack.top != 1) {
42                  return false;
43              }
44
45              memset(tempData, 0, MAX_EXPR_LEN * 2);
46              j = 0;
47          } else if (element == '+' || element == '-' || element == '*
                ' || element == '/') {
48              // 操作符入栈
49              char operator[2] = {0};
50              operator[0] = element;
51              Push(&operStack, operator);
52          }
53          else if (element == ')') {
54              // 右括号处理
55              char tmp[MAX_EXPR_LEN * 2] = {0};
56
57              if (dataStack.top - operStack.top != 1) {
58                  return false;
59              }
60
61              char* right = Pop(&dataStack);
62              char* left;
63
64              if (dataStack.top == -1) {
65                  // 如果操作数栈为空，直接添加左括号
66                  sprintf(tmp, "(%s)", right);
67                  Push(&dataStack, tmp);
68              } else {
69                  left = Pop(&dataStack);
70                  sprintf(tmp, "(%s%s%s)", left, operStack.data[
                        operStack.top--], right);
71                  Push(&dataStack, tmp);
72              }
73          } else {
```

```
74              return false;
75          }
76      }
77
78      // 处理剩余的操作数
79      while (dataStack.top > -1) {
80          char tmp[MAX_EXPR_LEN * 2] = {0};
81          char* right = Pop(&dataStack);
82          char* left;
83
84          if (dataStack.top == -1) {
85              strcpy(result, right);
86
87              // 检查是否还有剩余操作符
88              if (operStack.top != -1) {
89                  return false;
90              }
91
92              return true;
93          } else {
94              // 组装操作符和操作数
95              if (operStack.top == -1) {
96                  return false;
97              }
98
99              left = Pop(&dataStack);
100             sprintf(tmp, "%s%s%s", left, operStack.data[operStack.
                    top--], right);
101             Push(&dataStack, tmp);
102         }
103     }
104
105     return true;
106 }
```

3.3 主函数

主函数接收输入的字符串，传入补全表达式的函数中，若函数返回值为 true，则输出补全后的表达式；否则输出表达式有误的提示。

```
1  int main() {
2      printf("Please enter an expression without left parentheses.\n")
```

```
        ;

    char expression[MAX_EXPR_LEN] = {0};
    char result[MAX_EXPR_LEN * 2] = {0};

    gets(expression);

    bool success = completeParentheses(expression, result);

    if (success) {
        printf("The expression after completing parentheses: %s\n",
            result);
    } else {
        printf("Conversion failed. The input expression may be
            invalid.\n");
    }

    return 0;
}
```

4 调试分析报告

4.1 问题及解决方法

遇到的主要问题是表达式的存储方式和代码逻辑混乱。解决方法是将问题分解为不同条件下的子问题，再自上而下依次编写代码，同时考虑每一个分支对应的情况，是否能够包括所有的输入情况。

4.2 设计回顾与算法时空分析

算法设计主要使用了栈这个数据结构，程序的时间复杂度为 $O(n)$，空间复杂度上，程序使用了两个栈和两个字符串（分别存放输入和输出），在函数调用时还有额外的空间使用。

5 用户使用说明

将附录中的完整代码粘贴到 IDE 中运行即可。

6 测试结果

6.1 test 1

```
Please enter an expression without left parentheses.
1+2)*3)+4)
The expression after completing parentheses: (((1+2)*3)+4)
```

6.2 test 2

```
Please enter an expression without left parentheses.
1+2)*3+3*4
The expression after completing parentheses: (1+2)*3+3*4
```

6.3 test 3

```
Please enter an expression without left parentheses.
1+)2*3+4)
Conversion failed. The input expression may be invalid.
```

6.4 test 4

```
Please enter an expression without left parentheses.
1+2)))
The expression after completing parentheses: (((1+2)))
```

6.5 test 5

```
Please enter an expression without left parentheses.
1+*2)+3)
Conversion failed. The input expression may be invalid.
```

第二部分　判别回文字符串

7 需求分析

7.1 题目

正读和反读一样的字符串被称为回文字符串。编写程序，从键盘输入字符串（以 # 结尾），判断是否为回文字符串。

7.2 分析

可以将字符串入栈和入队列，再分别出栈和出队列。由于栈是后进先出，队列是先进先出，相当于将字符串倒着读和正着读，若前一半的字符都匹配，说明该字符串是回文串。
输入为一个以 # 结尾的字符串，若该字符串是回文串，则输出”Yes”，否则输出”No”。
输入输出示例：

```
//input
1234321#
//output
Yes
//input
12345421#
//output
No
```

8 概要设计

定义一个栈和一个队列，先初始化栈和队列，输入不为 # 时，将字符依次入栈和入队列，并记录元素数量。将元素依次出栈和出队列并比较，如果都相同，则证明是回文串。

9 详细设计

9.1 队列相关函数

包括队列初始化、入队、出队函数。

```
typedef struct queue{
    char s[MAX_LEN];
    int front;
    int rear;
}Queue;

Queue* init_Queue(){
    Queue *q= malloc(sizeof(struct queue));
    q->rear=-1;
    q->front=-1;
    return q;
}

void in(Queue* q,char c){
    q->s[++(q->rear)]=c;
```

```
}

char out(Queue* q){
    return q->s[++(q->front)];
}
```

9.2 栈相关函数

包括栈初始化、入栈、出栈函数。

```
typedef struct stack{
    char s[MAX_LEN];
    int top;
}Stack;

Stack *init_stack(){
    Stack* S=malloc(sizeof(struct stack));
    S->top=-1;
    return S;
}

void push(Stack* S,char c){
    S->s[++S->top]=c;
}

char pop(Stack* S){
    return S->s[S->top--];
}
```

9.3 主函数

实现输入、判断、输出功能。

```
int main(){
    char C;
    int len=0;
    Queue* q=init_Queue();
    Stack* S=init_stack();
    scanf("%c",&C);
    while(C!='#'){
        in(q,C);
        push(S,C);
        len++;
```

```
        scanf("%c",&C);
    }
    len=len/2;
    while(len--){
        if(out(q)!= pop(S)){
            printf("No\n");
            return 0;
        }
    }
    printf("Yes\n");
    return 0;
}
```

10 调试分析报告

10.1 时空分析

本程序主要使用了栈和队列数据结构，时间复杂度为 $O(n)$，空间上开辟了一个栈和一个队列，使用了一个字符变量和一个计数变量。

10.2 改进设想

字符串的长度为 n，则只需要判断前 $n/2$ 个元素即可。

11 用户使用说明

将代码在 IDE 中运行即可。

12 测试结果

12.1 test 1

```
123454321#
Yes
```

12.2 test 2

```
123432#
No
```

12.3 test 3

```
123$%^%$321#
Yes
```

12.4 test 4

```
1#
Yes
```

A 括号补全代码

```c
#include <stdio.h>
#include <string.h>

#define bool char
#define true 1
#define false 0
#define MAX_EXPR_LEN 50

typedef struct Stack {
    char data[MAX_EXPR_LEN][MAX_EXPR_LEN * 2];
    int top;
} Stack;

// 初始化栈
void initStack(Stack* s) {
    s->top = -1;
}

// 入栈操作
void Push(Stack* s, char* data) {
    s->top++;
    strcpy(s->data[s->top], data);
}

// 出栈操作
char* Pop(Stack* s) {
    return s->data[s->top--];
}
```

```
29
30 // 补全括号函数
31 bool completeParentheses(char* expression, char* result) {
32     int exprLength = strlen(expression);
33
34     if (exprLength >= MAX_EXPR_LEN) {
35         return false;
36     }
37
38     char tempData[MAX_EXPR_LEN * 2] = {0};
39     Stack dataStack, operStack;
40
41     initStack(&dataStack);
42     initStack(&operStack);
43
44     int j = 0;
45
46     // 遍历表达式
47     for (int i = 0; i < exprLength; i++) {
48         char element = expression[i];
49
50         if (element >= '0' && element <= '9') {
51             // 获取操作数
52             tempData[j++] = element;
53             i++;
54
55             while (i < exprLength) {
56                 char element1 = expression[i];
57
58                 if (element1 >= '0' && element1 <= '9') {
59                     tempData[j++] = element1;
60                     i++;
61                 } else {
62                     i--;
63                     break;
64                 }
65             }
66
67             Push(&dataStack, tempData);
68
69             // 栈顶元素判断
```

```
70              if (dataStack.top - operStack.top != 1) {
71                  return false;
72              }
73
74              memset(tempData, 0, MAX_EXPR_LEN * 2);
75              j = 0;
76          } else if (element == '+' || element == '-' || element == '*
                ' || element == '/') {
77              // 操作符入栈
78              char operator[2] = {0};
79              operator[0] = element;
80              Push(&operStack, operator);
81          }
82          else if (element == ')') {
83              // 右括号处理
84              char tmp[MAX_EXPR_LEN * 2] = {0};
85
86              if (dataStack.top - operStack.top != 1) {
87                  return false;
88              }
89
90              char* right = Pop(&dataStack);
91              char* left;
92
93              if (dataStack.top == -1) {
94                  // 如果操作数栈为空，直接添加左括号
95                  sprintf(tmp, "(%s)", right);
96                  Push(&dataStack, tmp);
97              } else {
98                  left = Pop(&dataStack);
99                  sprintf(tmp, "(%s%s%s)", left, operStack.data[
                        operStack.top--], right);
100                 Push(&dataStack, tmp);
101             }
102         } else {
103             return false;
104         }
105     }
106
107     // 处理剩余的操作数
108     while (dataStack.top > -1) {
```

```
109         char tmp[MAX_EXPR_LEN * 2] = {0};
110         char* right = Pop(&dataStack);
111         char* left;
112 
113         if (dataStack.top == -1) {
114             strcpy(result, right);
115 
116             // 检查是否还有剩余操作符
117             if (operStack.top != -1) {
118                 return false;
119             }
120 
121             return true;
122         } else {
123             // 组装操作符和操作数
124             if (operStack.top == -1) {
125                 return false;
126             }
127 
128             left = Pop(&dataStack);
129             sprintf(tmp, "%s%s%s", left, operStack.data[operStack.
                top--], right);
130             Push(&dataStack, tmp);
131         }
132     }
133 
134     return true;
135 }
136 
137 int main() {
138     printf("Please enter an expression without left parentheses.\n")
        ;
139 
140     char expression[MAX_EXPR_LEN] = {0};
141     char result[MAX_EXPR_LEN * 2] = {0};
142 
143     gets(expression);
144 
145     bool success = completeParentheses(expression, result);
146 
147     if (success) {
```

```
148         printf("The expression after completing parentheses: %s\n",
                result);
149     } else {
150         printf("Conversion failed. The input expression may be
                invalid.\n");
151     }
152
153     return 0;
154 }
```

B 回文字符串代码

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MAX_LEN 100
5  typedef struct queue{
6      char s[MAX_LEN];
7      int front;
8      int rear;
9  }Queue;
10
11 Queue* init_Queue(){
12     Queue *q= malloc(sizeof(struct queue));
13     q->rear=-1;
14     q->front=-1;
15     return q;
16 }
17
18 void in(Queue* q,char c){
19     q->s[++(q->rear)]=c;
20 }
21
22 char out(Queue* q){
23     return q->s[++(q->front)];
24 }
25
26 typedef struct stack{
27     char s[MAX_LEN];
28     int top;
```

```
}Stack;

Stack *init_stack(){
    Stack* S=malloc(sizeof(struct stack));
    S->top=-1;
    return S;
}

void push(Stack* S,char c){
    S->s[++S->top]=c;
}

char pop(Stack* S){
    return S->s[S->top--];
}
int main(){
    char C;
    int len=0;
    Queue* q=init_Queue();
    Stack* S=init_stack();
    scanf("%c",&C);
    while(C!='#'){
        in(q,C);
        push(S,C);
        len++;
        scanf("%c",&C);
    }
    len=len/2;
    while(len--){
        if(out(q)!= pop(S)){
            printf("No\n");
            return 0;
        }
    }
    printf("Yes\n");
    return 0;
}
```