

### ▸ 程式解說

#### ● Infix to Postfix - 中序式後序式轉

我們可以用兩種方法來達到此目標，一種是建立二元樹的方式，接著再用後序式走訪，即可將算式轉換成後序式，但此方法相較於直接做運算來說，過於複雜，因此我們會採用由Edsger W. Dijkstra所設計的「調度場演算法」。

1. 由左至右走訪中序式
2. 若遇到數字則直接輸出
3. 當遇到“(”、“[”、“{”，push進入Stack
4. 若遇到“)”、“]”、“}”，則將Stack內的運算子pop出來，直到遇到“(”、“[”、“{”
5. 遇到運算子時，將Stack中的運算子做輸出，直到<sup>1</sup>.Stack最上面的運算子之優先權小於現在所走訪到的運算子或是<sup>2</sup>.Stack已為空時，再將自己push進入
6. 重複1.~5.的步驟，直到走訪完畢
7. 最後將Stack內的運算子全部pop出來

Operator	Precedence
^	3
*	2
/	2
+	1
-	1

### 運算子優先權

左邊的表格可以算是最基本的優先權，數字越大，代表優先權越大，但因為這裡加入了括號，因此要加以討論。

加入括號後，我們可以發現，當在Step.3時，括號的優先權要是最大的，擁有最高的優先權，可以直接進入Stack，但Step.5時，要push進入Stack時，只與堆疊內的運算子之優先權做比較。另一種說法也就是，若Stack最上方是括號，那麼就可以直接壓在括號上面，而這時括號則是最低的優先權。

但，上述的內容發生矛盾，因此我們要分開討論。

- Precedence 優先級 - ISP and ICP

```
1 int ISP(char c){ // ISP -> In Stack Priority
2     switch(c){
3         case '^':
4             return 3;
5         case '*':
6         case '/':
7             return 2;
8         case '+':
9         case '-':
10            return 1;
11        case '(':
12        case '[':
13        case '{':
14            return 0;
15    }
16
17    return -1;
18 }
```

```
1 int ICP(char c){ // ICP -> Incoming Priority
2     switch(c){
3         case '(':
4             return 6;
5         case '[':
6             return 5;
7         case '{':
8             return 4;
9         case '^':
10            return 3;
11        case '*':
12        case '/':
13            return 2;
14        case '+':
15        case '-':
16            return 1;
17    }
18
19    return -1;
20 }
```

經由上述的討論，我們發現會遇到兩種情況，一種是已經在Stack裡，而另一種是正要進入Stack，因此我們可以這兩種優先權的分級。

- In Stack Priority (ISP): 當“(”在裡面時，優先權最低，所有的運算子都可以push進Stack裡，壓在“(”上面
- Incoming Priority (ICP): 當“(”在外面時，優先權最高，可以push進堆疊壓在任何運算子上面

## ● Character Checker - 字元判斷

```
1  bool isDigit(char c){
2      if(c >= '0' && c <= '9')
3          return true;
4      else
5          return false;
6  }
7
8  bool isOperator(char c){
9      if(c == '+' || c == '-' || c == '*' || c == '/' || c == '^')
10         return true;
11     else
12         return false;
13 }
14
15 bool isParentheses(char c){
16     if(c == '(' || c == '[' || c == '{' || c == ')' || c == ']' || c == '}')
17         return true;
18     else
19         return false;
20 }
```

- bool isDigit(char c) => 確認此字元是不是數字
- bool isOperator(char c) => 確認此字元是不是運算子
- bool isParentheses(char c) => 確認此字元是不是括號

## ● Brackets Checker - 括號判斷

```
1  bool checkBrackets(){
2      if(ValidParentheses.size() > 1){ // check if there are more than one brackets
3          char up = ValidParentheses.top(); ValidParentheses.pop();
4          char dn = ValidParentheses.top(); ValidParentheses.pop();
5          if((up - '0') > (dn - '0')){ // check if the up bracket is bigger than the down bracket
6              return false;
7          }
8          else{
9              ValidParentheses.push(dn);
10             ValidParentheses.push(up);
11         }
12     }
13
14     return true;
15 }
```

- 判斷該中序式是不是符合大括號包中括號包小括號
- up是對上面的括號，而dn是第二個，因此若up的ascii-code大於dn的ascii-code也就發生了較小括號包住了較大的括號，因此為錯誤，return false，若沒有發生則目前是合法的，因此再放入stack裡

- 基本運作及變數介紹

```
1  int main(){
2      string str;
3
4      while(getline(cin, str)){
5          stack <char> s;
6
7          string ans = ""; // postfix
8          string clear_str = ""; // infix
9
10         int val = 0;
11         bool isValid = true;
12
13         int num_cnt = 0; // numbers of digits
14         int op_cnt = 0; // numbers of operators
15
16         for(int i=0; i<str.length(); i++){
17             // scan the string...
18         }
19
20         // ...
21     }
22 }
```

先做基本輸入，而我們這裡使用getline()，讓他可以接收包含空白字串

- stack <char> s => 存入運算子的堆疊
- string str => 輸入的中序式字串
- string clear\_str => 篩選掉一些不需要的字元後，所存入的中序式字串
- int val => 如果是數字的話，將值暫時存入這裡
- bool isValid => 確認這個中序式是不是合法的
- int num\_cnt => 紀錄有幾個數字
- int op\_cnt => 紀錄有幾個運算子

## ● 篩選字元

```
1 while(getline(cin, str)){
2     // ...
3
4     for(int i=0; i<str.length(); i++){
5         if(!isDigit(str[i]) && !isOperator(str[i]) && !isParentheses(str[i])){ // if str[i] is not a digit, operator, or parentheses
6             continue;
7         }
8
9         if(!isDigit(str[i])){
10            clear_str += str[i];
11            clear_str += ' ';
12        }
13
14        if(!checkBrackets()){ // check brackets are valid
15            isValid = false;
16            break;
17        }
18
19        // ...
20    }
21
22    // ...
23 }
```

我們將除了數字、運算子、括號等之外的字元做過濾，若不是數字先丟入clear\_str，而若為數字則處理完後再放入clear\_str裡面

由於題目設定為“字串中夾帶有+ - \* / ^ 0 1 2 3 4 5 6 7 8 9 ( ) [ ] { } 以外的符號直接輸出ERROR”，因此這裡已做更改

## ● 讀取數字

```
1 while(getline(cin, str)){
2     // ...
3
4     if(isDigit(str[i])){ // if str[i] is a digit
5         val += (str[i] - '0');
6
7         int index = i+1;
8         while(index != str.length() && isDigit(str[index])){
9             val = val * 10 + (str[index] - '0');
10            index++;
11        }
12
13        if(index != str.length()){
14            if(str[index] == '(' || str[index] == '[' || str[index] == '{'){
15                isValid = false;
16                break;
17            }
18        }
19
20        clear_str += to_string(val);
21        clear_str += ' ';
22
23        num_cnt++;
24        ans += to_string(val);
25        ans += ' ';
26        val = 0;
27        i = index - 1;
28    }
29    else if ...
30
31 }
```

若str[i]為數字，我們利用index和迴圈直將將這個數字一次取出來，並放入clear\_str，最後若str[index]為括號，那則為不合法，直接將isValid做記號，跳出迴圈，若不是則直接存入ans並且加上空格加上區隔，進入下個迭代

- 若為右括號

```
1 while(getline(cin, str)){
2     // ...
3
4     else if(str[i] == ')' || str[i] == ']' || str[i] == '}'){ // if str[i] is a right parentheses
5         if(ValidParentheses.empty()){
6             isValid = false;
7             break;
8         }
9         else{
10            char top = ValidParentheses.top(); ValidParentheses.pop();
11            if((top == '(' && str[i] != ')') || (top == '[' && str[i] != ']') || (top == '{' && str[i] != '}')){ // check parentheses are valid
12                isValid = false;
13                break;
14            }
15            else{
16                while(s.top() != '(' && s.top() != '[' && s.top() != '{'){ // pop all operators in stack
17                    ans += s.top();
18                    ans += ' ';
19                    op_cnt++;
20                    s.pop();
21                }
22                s.pop(); // pop left parentheses
23            }
24        }
25    }
26    // ...
27 }
28 }
```

若是為右括號的情況，我們先看ValidParentheses是不是空的，若是空的或是括號沒有對應到，代表這個中序式不合法，因此將isValid做記號，跳出迴圈，其他情況代表合法，將Stack內的運算子pop出來，直到出現左括號，最後再將左括號pop出來

- 若為左括號或運算子 - Case1: stack is empty or ICP > ISP

```
1 while(getline(cin, str)){
2     // ...
3
4     else if(s.empty() || (ICP(str[i]) > ISP(s.top()))){ // if str[i] is a left parentheses or operator with higher precedence
5         s.push(str[i]);
6
7         if(str[i] == '(' || str[i] == '[' || str[i] == '{'){ // if str[i] is a left parentheses
8             ValidParentheses.push(str[i]);
9         }
10    }
11    // ...
12 }
13 }
```

當現在的字元為左括號或是運算子時，會出現現在字元的優先權大於堆疊最上面的元素的優先權或者其他，而我這裡先討論前者，當大於時直接將此字元push進堆疊，而若這個字元是左括號時則也push進入ValidParentheses裡

- 若為左括號或運算子 - Case2: ICP <= ISP

```
1 while(getline(cin, str)){
2     // ...
3
4     else{
5         while(!s.empty() && (ICP(str[i]) <= ISP(s.top()))){ // if str[i] is a operator with lower precedence
6             ans += s.top();
7             ans += ' ';
8             op_cnt++;
9             s.pop();
10        }
11        s.push(str[i]); // push str[i] to stack
12
13        if(str[i] == '(' || str[i] == '[' || str[i] == '{'){
14            ValidParentheses.push(str[i]);
15        }
16    }
17
18    // ...
19 }
```

繼上一個case後，現在來討論另個案例，若於小於等於時，我們則是將堆疊裡的運算子先pop出來，直到堆疊裡的運算子之優先權小於現在運算子的優先權，最後在將目前的運算子push進堆疊裡，而若這個字元是左括號時則也push進入ValidParentheses裡

- 最後輸出和後序式運算

```
1 while(getline(cin, str)){
2     // ...
3
4     while(!s.empty()){ // pop all operators in stack
5         ans += s.top();
6         ans += ' ';
7         op_cnt++;
8         s.pop();
9     }
10
11     if(isValid && checkBrackets() && ValidParentheses.empty() && num_cnt == op_cnt + 1){ // check infix is valid
12         cout << "Infix: " << clear_str << endl;
13         postfix_operator(ans); // calculate postfix
14         // output_oj(ans);
15         // cout << ans << endl;
16     }
17     else
18         cout << "ERROR" << endl;
19
20     while(!ValidParentheses.empty()){ // clear ValidParentheses
21         ValidParentheses.pop();
22     }
23
24
25     cout << endl;
26 }
```

由左至右走訪完畢後，我們將還在堆疊裡面的運算子pop出來，接者經由確認此中序式是合法的後，輸出經由過濾過的中序式，最後進入後序式運算，結束後再將ValidParentheses做清空

輸出的部分已根據測資和題目規定做更改

## ● 後序式運算

```
1 void postfix_operator(string str){
2     stack<int> nums; // store the numbers
3
4     for(int i=0; i<str.length(); i++){
5         if(isDigit(str[i])){ // if the character is a number
6             int val = 0;
7             while(isDigit(str[i])){
8                 val = val * 10 + (str[i] - '0');
9                 i++;
10            }
11
12            nums.push(val);
13        }
14        else if(str[i] != ' '){ // if the character is an operator
15            int b = nums.top(); nums.pop(); // pop the top two numbers
16            int a = nums.top(); nums.pop();
17
18            switch(str[i]){
19                case '+':
20                    nums.push(a + b);
21                    break;
22                case '-':
23                    nums.push(a - b);
24                    break;
25                case '*':
26                    nums.push(a * b);
27                    break;
28                case '/':
29                    nums.push(a / b);
30                    break;
31                case '^':
32                    nums.push(pow(a, b));
33                    break;
34            }
35        }
36    }
37
38    cout << "Postfix: " << str << endl;
39    cout << "Ans: " << nums.top() << endl;
40 }
```

L2: 先設立一個堆疊，用來存放數字

L4: 由左至右走訪後序式

L5: 若是數字，則如同main裡面的做法一樣，將數字取過來，並push進堆疊

L14: 若不是數字也不是空格，那必定是運算子，進行運算

L15: 將最上面的數字取出，設為b，並做pop

L16: 將最上面的數字取出，設為a，並做pop

L18: 進入運算，依照運算子，最後在將答案push進堆疊裡

L38: 輸出後序式字串

L39: 取出top，輸出運算結果，回到主函式

輸出結果已根據題目做更改



## ▶ 程式輸出結果

```
~/De/code/L/C_Programming/f/2/DS_HW2-Arithmetic > main 13 72
>$ ./test
( ( 2 * ( 3 - 4 ) ) + ( 12 / 4 jiojeoiwjexo ) )
Infix: ( ( 2 * ( 3 - 4 ) ) + ( 12 / 4 ) )
Postfix: 2 3 4 - * 12 4 / +
Ans: 1

( ( 2 * ( 3 - 4 ) ) + ( 12 / 4 ) )
Infix: ( ( 2 * ( 3 - 4 ) ) + ( 12 / 4 ) )
Postfix: 2 3 4 - * 12 4 / +
Ans: 1

((2*(3-4))+(12/4))
Infix: ( ( 2 * ( 3 - 4 ) ) + ( 12 / 4 ) )
Postfix: 2 3 4 - * 12 4 / +
Ans: 1

(10+3)*5-6
Infix: ( 10 + 3 ) * 5 - 6
Postfix: 10 3 + 5 * 6 -
Ans: 59

2^[4*(5-(2+3))]
Infix: 2 ^ [ 4 * ( 5 - ( 2 + 3 ) ) ]
Postfix: 2 4 5 2 3 + - * ^
Ans: 1

3+(2*[1+4])
ERROR

3+(2*[1+4])
ERROR

3 * 2^3
Infix: 3 * 2 ^ 3
Postfix: 3 2 3 ^ *
Ans: 24

{[5-4]/8}*6
Infix: { [ 5 - 4 ] / 8 } * 6
Postfix: 5 4 - 8 / 6 *
Ans: 0.75

12-78-eva
ERROR
```

- 我的版本可以過濾掉不相干的字元，並且做輸出中序式和後序式，最後再做運算

```
~/De/code/L/C_Programming/f/2/DS_HW2-Arithmetic > main 14 75
>$ ./test
(10+3)*5-6
59
2^[4*(5-(2+3))]
1
3+(2*[1+4])
ERROR
2(1+3)
ERROR
1+ 2
ERROR
( ( 2 * ( 3 - 4 ) ) + ( 12 / 4 jiojeoiwjexo ) )
ERROR
((2*(3-4))+(12/4))
1
2*(3-4)+12/4
1
█
```

- 這是題目所要求的版本

## ▸ 心得

一開始拿到這個題目的時候，我以為會像之前寫的那種，有小數、奇怪字元、空格，要做過濾和在處理上要修改一下，而寫完後和朋友的對照一下才發現，原來沒那麼複雜，只是做基本的轉換並做後序運算，但思考的過程的確很有趣，要考慮各種奇怪的特例，並且做不斷不斷地修正，當然，這次也要教導朋友寫作業，但透過教學和討論，也看到不同的思考與解法，而透過報告也可以使我們更明白，自己的程式在做什麼，並了解哪些可以再做優化，這個作業使我受益良多。

## ▸ 補充說明

iLearn上繳交的程式碼是為了符合題目的目標所改的，而報告上的輸出則是放在GitHub上，歡迎助教與老師參閱，謝謝。

URL: [https://github.com/alecwu44743/c\\_learning/blob/main/fcu\\_cs/22fall\\_Data\\_Structures/DS\\_HW2-Arithmetic/Infix2Postfix\\_Arithmetic.cpp](https://github.com/alecwu44743/c_learning/blob/main/fcu_cs/22fall_Data_Structures/DS_HW2-Arithmetic/Infix2Postfix_Arithmetic.cpp)