Prefix Evaluation

In, Prefix evaluation , the conversion from infix to prefix evaluation will remain same.

Now if we consider a infix expression :

$\rightarrow (a + b)\hat{}(c - d)/e$

can be observed through a stack implementation table:

Stack	*Input*	*Output*
Empty	$(a + b)\hat{}(c - d)/e$	*Nothing*
Empty	$(a + b)\hat{}(c - d)/$	e
$/$	$(a + b)\hat{}(c - d)$	e
$)/$	$(a + b)\hat{}(c - d$	e
$)/$	$(a + b)\hat{}(c -$	ed
$-)/$	$(a + b)\hat{}(c$	ed
$-)/$	$(a + b)\hat{}($	edc
$/$	$(a + b)\hat{}$	$edc -$
$\hat{}/$	$(a + b)$	$edc -$
$)\hat{}/$	$(a + b$	$edc -$
$)\hat{}/$	$(a +$	$edc - b$
$+)\hat{}/$	$(a$	$edc - b$
$+)\hat{}/$	$($	$edc - ba$
$\hat{}/$	$($	$edc - ba +$
$/$	*Empty*	$edc - ba + \hat{}$
Empty	*Empty*	$edc - ba + \hat{}/$

$Now\ reverse\ edc - ba + \hat{}/\ \ to\ get\ prefix\ \ expression$:

$/\hat{} + ab - cde.$

$Similarly, if\ we\ have\ a\ infix\ notation$:

$(2 + 3) * (2 + 3)/5$

$Now\ we\ get\ the\ converted\ postfix\ expression\ as$:

$\rightarrow * +23/+235$

$Stack$	$Input$	$Output$
$Empty$	$(2 + 3) * (2 + 3)/5$	$Nothing$
$Empty$	$(2 + 3) * (2 + 3)/$	5
$/$	$(2 + 3) * (2 + 3)$	5
$)/$	$(2 + 3) * (2 + 3$	5
$)/$	$(2 + 3) * (2 +$	53
$+)/$	$(2 + 3) * (2$	53
$+)/$	$(2 + 3) * ($	532
$/$	$(2 + 3) *$	$532 +$
$*/$	$(2 + 3)$	$532 +$
$) */$	$(2 + 3$	$532 +$
$) */$	$(2 +$	$532 + 3$
$+) */$	$(2$	$532 + 3$
$+) */$	$($	$532 + 32$
$*/$	$($	$532 + 32 +$
$/$	$Empty$	$532 + 32 +*$
$Empty$	$Empty$	$532 + 32 +*/$

Reversing the $532 + 32 +*/$ ***to*** $* +23/+235$.

Now what will be the result : $5 * 5/5 = \frac{25}{5} = 5$

Lets take an example to evaluate : $(2 + 3) * (2 + 6)$

prefix expression: $* +23 + 26$

Here we will start from the end of the expression.

Lets evaluate it through stack :

Push (6),

Push(2),

Now we get $+$ ***hence*** : ***Add***$(2, 3) = 2 + 6 = 8$

Like what it happens in Stack in memory:

6
2
Data Register: $+$

Then it pop out , 6 ***and*** 2 ***and send ADD***$(6, 2)$ ***to Processor to process***.

$Similarly\ we\ have\ the\ prefix\ expression\colon\ *+23+26$

$if\ it\ is\ not\ operator\ i.e.operand\ Push\ it\ to\ the\ stack.$

```
int prefixEvaluation(char *prefix)
{
    Stack st;
    int len = strlen(prefix);
    create(&st, len);

    int i = len - 1;

    int r, op1, op2;

    while (i >= 0)
    {
        if (isOperand(prefix[i]) == 1)
        {
            push(&st, prefix[i] - '0');
        }
        else
        {
         ….
         }
         ….
        }
       i--;
     }
}
```

1. *The above function takes the prefix expression.*
2. *Create another stack again after prefix conversion,*
3. *Till the first character, the characters will be taken inside the loop.*

First is 6 *and* 2 *, both are operand and both will get pushed inside Stack.*

ASCII value of 0 *is* 48 *. Now we see that we will push integer values not characters, hence* 0 *is* 48 *,* 1 *is* 49 *,* 2 *is* 50 *and so on.*

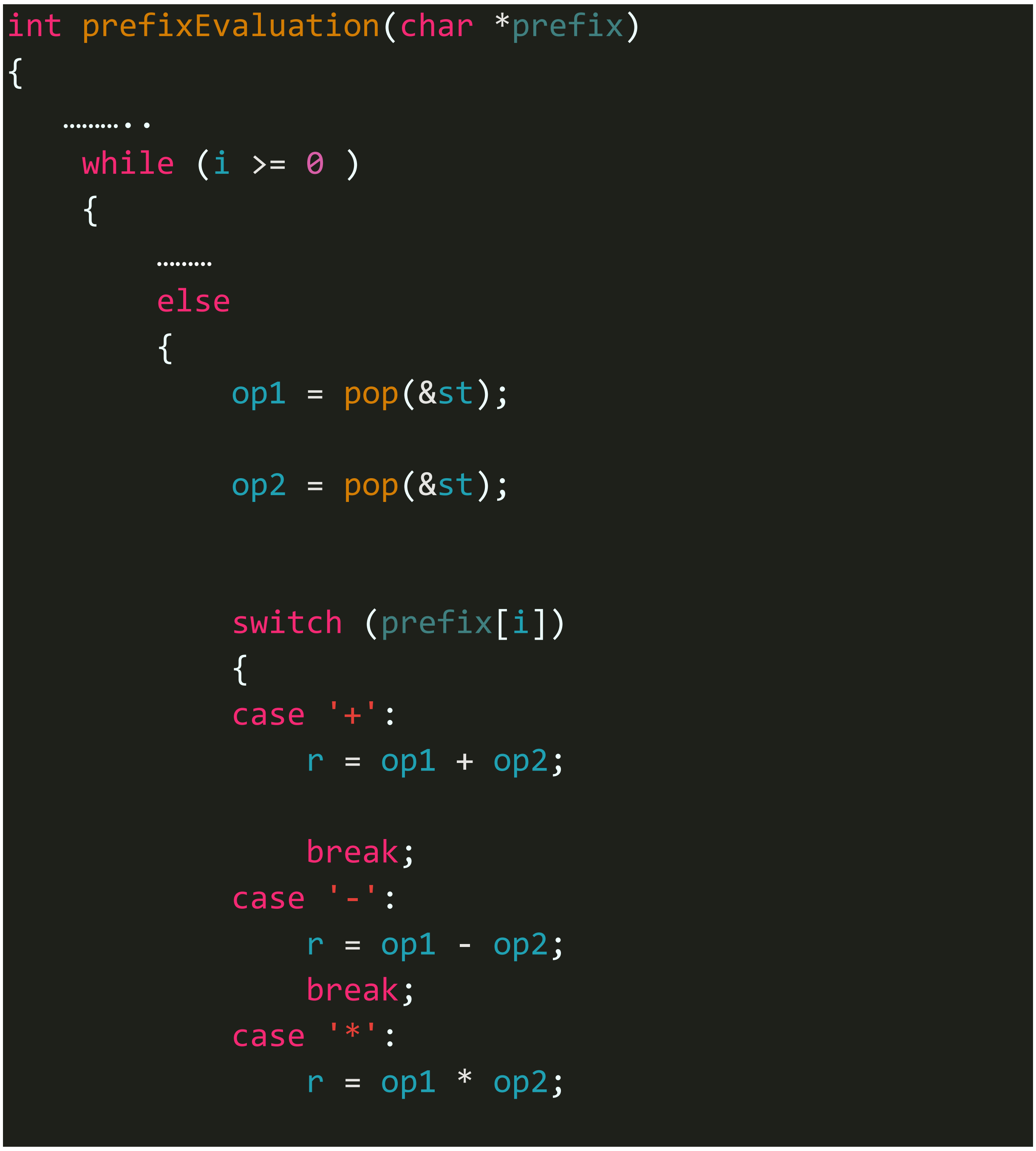
If we do : $50 - 48$ *i.e.* (*ASCII of* 2 − *ASCII of* 0) *will be* 2.
Similarly, $54 - 48$ *i.e.* (*ASCII of* 6 − *ASCII of* 0) *will be* 6.

Hence,
1. *push*(`2` − `0` = 2)
2. *push* (`6` − `0` = 6)

6
2

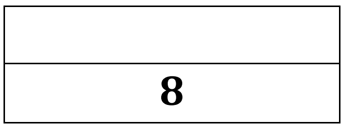
STACK

4. Now we have` + `operator, hence we pop out 6 first and then 2 and it adds the two operands and after the result we get , push it into the stack.



```c
int prefixEvaluation(char *prefix)
{
   ……….. .
    while (i >= 0 )
    {
        ………
        else
        {
            op1 = pop(&st);

            op2 = pop(&st);


            switch (prefix[i])
            {
            case '+':
                r = op1 + op2;

                break;
            case '-':
                r = op1 - op2;
                break;
            case '*':
                r = op1 * op2;
                break;
            case '/':
                r = op1 / op2;

                break;
            case '^':
                r = pow(op1, op2);
                break;
            case '%':
                r = op1 % op2;
                break;
            }
            push(&st, r);
        }
        i--;
    }

    …..
}
```

$6 + 2 = 8$ ***and we push***(8) ***into the stack.***



STACK

5. *The process continues i.e. push* 3 *and* 2 *into the stack,*

2
3
8

STACK

Then pop(2, 3) *from stack and Add*(2, 3) = 5 *and push*(5) *into the stack.*

5
8

STACK

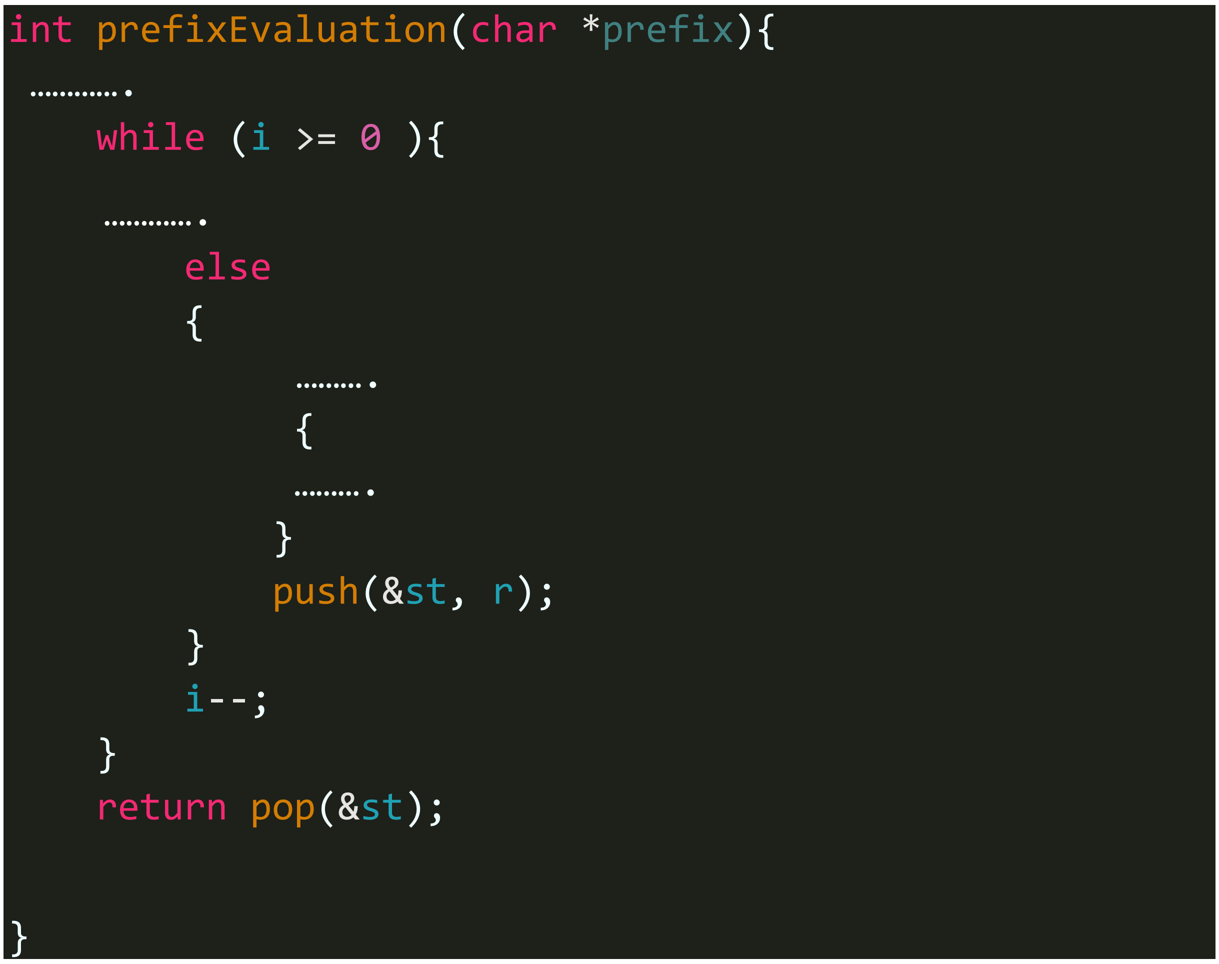
6. *Now we get* ` ∗ ` *operator , hence both operands i.e.* 8 *and* 5 *will get pop out from the stack and get multiplied with output* = 40.

The output 40 *will get pushed into the stack.*

40

STACK

7. *Now, pop out 40 as output*:



```
int prefixEvaluation(char *prefix){
 ………….
    while (i >= 0 ){

     ………….
        else
        {
             ……….
             {
             ……….
            }
            push(&st, r);
        }
        i--;
    }
    return pop(&st);


}
```

<u>This process is known as Prefix Evaluation.</u>

Time Complexity of

PreFix Evaluation

```
int prefixEvaluation(char *prefix)
{
    Stack st; → O(1)
    int len = strlen(prefix);→ O(n)
    create(&st, len);→ O(n)

    int i = len - 1;→ O(1)

    int r, op1, op2;→ O(1)

    while (i >= 0 )→ O(n)
    {
        if (isOperand(prefix[i]) == 1)
        {
            push(&st, prefix[i] - '0');→ O(1)
        }
        else
        {
            op1 = pop(&st);→ O(1)

            op2 = pop(&st); );→ O(1)
            switch (prefix[i])→ O(1)
            {
            case '+':
                r = op1 + op2;

                break;
            case '-':
                r = op1 - op2;
                break;
            case '*':
                r = op1 * op2;

                break;
            case '/':
                r = op1 / op2;

                break;
            case '^':
                r = pow(op1, op2);
                break;
            case '%':
                r = op1 % op2;
                break;
            }
            push(&st, r);→ O(1)
        }
        i--;
    }

    return pop(&st);→ O(1)
}
```

1. ***Creating object of stack `Stack st` takes $O(1)$ time.***

2. ***Calculating the length of the prefix expression strlen(Prefix): $O(n)$, where n is the length of the Prefix Expression.***

3. ***Creation of the stack of n length of prefix expression: create(&st, len) $\rightarrow O(n)$.***

4. ***int $i = len - 1$; $\rightarrow O(1)$ constant time.***

5. ***Declaration of variables : int op1, op2, r takes $O(1)$ time complexity.***

6. *While loop runs from last to 0 takes $O(n)$time complexity.*

7. ***Push and Pop occurs $O(1)$ time at each operation. Total push pop occurs $O(n)$ as we scan the postfix expression from first to last.***

8. ***Performing arithmetic operations: The switch statement performs arithmetic operations such as addition, subtraction, multiplication, division, exponentiation, and modulo. These operations also take constant time, $O(1)$time at each operation. As at each specific switch , each case runs only one time.When it runs upto the length of the expression it takes $O(n)$ time.That is cases operates $O(n)$ times in switch-case.***

9.***And returning the pop operation also takes $O(1)$ time.***

Hence, $O(1) + O(n) + O(n) + O(1) + O(1) + O(n) + O(1) = O(n)$ time complexity.

$O(n)$ *where, n is the length of the prefix string.*

<u>Space Complexity</u>

Push operation in stack takes $O(n) = O(n)$ complexity.

That is `n` is the auxiliary space taken during push operation.

Hence Space Complexity $= O(n)$ complexity.