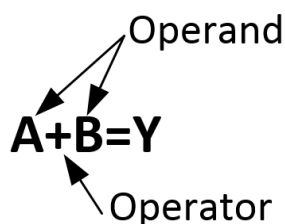


2.3 นิพจน์อินฟิกและโพสฟิก

การแปลงนิพจน์อินฟิก (Infix) ให้เป็นนิพจน์โพสฟิก (Postfix) เป็นกระบวนการสำคัญต่อการออกแบบโปรแกรมภาษามาก ปกติแล้วคอมไพเลอร์ (Compiler) จะไม่สามารถ สร้างรหัสหรือชุดคำสั่งจากนิพจน์คณิตศาสตร์ที่ป้อนเข้าเครื่อง เช่น $A+B$ ตัวคอมไพเลอร์ต้อง แปลงนิพจน์นี้เป็น $AB+$ เสียก่อนแล้วค่อยแปลงเป็นชุดคำสั่งของเครื่องอีกต่อหนึ่ง ข้อความที่อยู่ทางซ้ายมือของเครื่องหมายเท่ากับ (=) ซึ่งอาจจะเป็นข้อมูลคงที่ หรือตัวแปรเพียงตัวเดียว ก็ได้ หรืออาจเป็นข้อความเกี่ยวกับการคำนวณก็ได้

ลักษณะนิพจน์อินฟิก (Infix Notation)

นิพจน์อินฟิกตัวโอเปอเรเตอร์ (Operator) จะอยู่ตรงกลางระหว่างโอเปอเรนด์ (Operand) ทั้งสองตัวที่เป็นกันตามปกติทั่วไป



ลักษณะนิพจน์โพสฟิก (Postfix Notation)

นิพจน์โพสฟิกตัวโอเปอเรเตอร์ (Operator) จะอยู่ข้างหลังของโอเปอเรนด์ (Operand) ทั้งสองตัว

Infix notation	Postfix notation
$A + B$	$A B +$
$A - B$	$A B -$
$A * B$	$A B *$
A / B	$A B /$
$A \wedge B$	$A B \wedge$

รูปที่ 2.22 นิพจน์อินฟิกกับโพสฟิก

2.3.1 การแปลงนิพจน์อินฟิกเป็นโพลฟิก

การแปลงนิพจน์ อินฟิกเป็นโพลฟิก นั้นต้องใช้กลไกสแตกเป็นตัวช่วย นอกจากนี้ แล้วยังต้องนิยามความสำคัญหรือลำดับก่อนหลังของโอเปอเรเตอร์ (Operator) ทาง คณิตศาสตร์โดยจะเริ่มโดยพิจารณาเครื่องหมาย 5 ตัว คือ + (บวก) , - (ลบ) , * (คูณ) , / (หาร) , 1 (ยกกำลัง) ซึ่งจะมีค่านัยสำคัญต่างกันไปตามตารางต่อไปนี้

เครื่องหมาย (Operator)	Precedence เมื่ออยู่ที่อินพุต	Precedence เมื่ออยู่ในสแตก
\wedge , $**$	4	3
$*$, $/$	2	2
$+$, $-$ (binary operator)	1	1
$($	4	0

รูปที่ 2.23 แสดงค่านัยสำคัญของเครื่องหมาย

ลำดับการคำนวณ

1. ทำการคำนวณซ้ายไปขวาเมื่อมีนัยสำคัญเท่ากัน
2. ถ้าไม่มีวงเล็บ ทำการคำนวณส่วนที่มีเครื่องหมายมีนัยสำคัญมากที่สุดก่อนแล้วจึงทำส่วนต่อไป
3. ถ้ามีวงเล็บทำการคำนวณส่วนที่อยู่ภายในวงเล็บจนหมดก่อนตามหลักการข้อที่ 1 และ 2 แล้วจึงทำส่วนต่อไป

วิธีการแปลงนิพจน์อินฟิกเป็นโพลฟิก

การแปลงอินฟิกเป็นโพลฟิก อาศัยการพิจารณานิพจน์อินฟิก ทีละตัวจากไปขวามืออยู่ 4 ขั้นตอนดังนี้

1. ถ้าอินพุตเป็น Operand ให้นำไปไว้ที่เอาต์พุต
2. ถ้าอินพุตเป็น Operator ให้ทำดังนี้
 - 2.1 สแตกว่างอยู่ นำ Operator ตัวนั้นเข้าสู่สแตกตัวที่หนึ่ง
 - 2.2 ถ้าสแตกไม่ว่าง ให้นำ Operator ที่เป็นอินพุตไปเทียบความสำคัญกับ Operator ที่อยู่ในสแตกถ้าตัวเข้ามาใหม่มีความสำคัญน้อยกว่าหรือเท่ากับ ก็ให้นำเอา Operator ที่อยู่ในสแตกบนสุดออกไปไว้เอาต์พุต แล้ว นำ Operator ที่

อินพุตไปเทียบความสำคัญกับ Operator ที่อยู่ในสแตกถัดไป แล้วก็ใช้หลักการ ทำ
จนกระทั่ง

- Operator ที่เป็นอินพุตมีความสำคัญมากกว่า Operator ที่อยู่ใน
สแตก

- ทำจนกระทั่งสแตกว่าง

- ทำจนกว่าจะพบเครื่องหมายวงเล็บเปิด “ (”

3. ถ้าอินพุตเป็นเครื่องหมายวงเล็บเปิด “ (” ก็ให้ PUSH ลงสู่สแตก แต่ถ้า อินพุตเป็น
เครื่องหมายวงเล็บปิด “) ” ให้ POP ข้อมูลในสแตกจนกว่าจะพบเครื่องหมายวงเล็บเปิด “ (” แล้ว
ทิ้งทั้งวงเล็บเปิดและปิด

4. ถ้าอินพุตหมด ให้ POP สแตกเอา Operator ออกมาที่เอาต์พุตจนหมด

ตัวอย่าง จงแปลงนิพจน์ $A+B+C-D$ ไปเป็นนิพจน์โพสฟิก

ข้อมูลเข้า(I/P)	โอเปอเรเตอร์สแตก	นิพจน์ Postfix (O/P)
A	NULL	A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
-	-	ABC*+
D	-	ABC*+D
NULL	NULL	ABC*+D-

รูปที่ 2.24 แสดงผลที่ได้จากการแปลงนิพจน์

ตัวอย่าง จงแปลงนิพจน์ $A+(B-(C+D))*E A^2$ ไปเป็นนิพจน์โพสฟิก

ข้อมูลเข้า(I/P)	โอเปอเรเตอร์สแตก	นิพจน์ Postfix (O/P)
A		A
+	+	A
(+(A
B	+(AB
-	+(-	AB
(+(-(AB
C	+(-(ABC
+	+(-(+	ABC
D	+(-(+	ABCD
)	+(-	ABCD+
)	+	ABCD+-
*	+*	ABCD+-
E	+*	ABCD+-E
^	+*^	ABCD+-E
2	+*^	ABCD+-E2
NULL		ABCD+-E2^*+

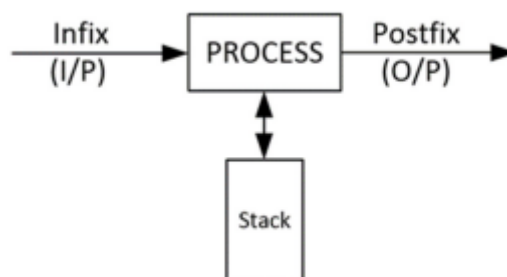
รูปที่ 2.25 แสดงผลที่ได้จากการแปลงนิพจน์

ตัวอย่าง จงแปลง $A+B*(CAD*E/F)-G$ ไปเป็นนิพจน์โพสฟิก

ข้อมูลเข้า(I/P)	โอเปอเรเตอร์สแตก	นิพจน์ Postfix (O/P)
A		A
+	+	A
(+(A
B	+(AB
-	+(-	AB
(+(-(AB
C	+(-(ABC
+	+(-(+	ABC
D	+(-(+	ABCD
)	+(-	ABCD+
)	+	ABCD+-
*	+*	ABCD+-
E	+*	ABCD+-E
^	+*^	ABCD+-E
2	+*^	ABCD+-E2
NULL		ABCD+-E2^*+

รูปที่ 2.26 แสดงผลที่ได้จากการแปลงนิพจน์ในตัวอย่างที่ 3

สรุปขั้นตอนการแปลงนิพจน์อินฟิก ไปเป็นนิพจน์โพสฟิก มีกระบวนการดังแสดงใน รูป 2.22



รูปที่ 2.27 แสดงไดอะแกรมการแปลงนิพจน์อินฟิกเป็นโพสฟิก

การวางโครงสร้างโปรแกรมแปลงนิพจน์อินฟิกเป็นโอสฟิก

นิพจน์อินฟิกจะกำหนดค่าไว้ในตัวแปรอาร์เรย์ชนิด Character เพื่อความสะดวกในการแยกมาพิจารณาเป็นอินพุตทีละตัว และทำการเตรียมฟังก์ชันการทำงานของสแตก (Push และ Pop) ในการเก็บค่าโอเปอเรเตอร์ จากนั้นก็ทำการนำข้อมูลเข้าสู่กระบวนการ แปลงทีละตัวตามหลักการที่ระบุไว้ ระหว่างนี้จะมีการเก็บค่าโอเปอเรเตอร์ไว้ในสแตกชั่วคราว เพื่อรอการประมวลผล เนื่องจากค่านัยสำคัญของโอเปอเรเตอร์เมื่อเป็นอินพุต กับเมื่ออยู่ในสแตกบางตัวจะมีค่าไม่เท่ากัน จึงสร้างโมดูลการกำหนดค่านัยสำคัญของโอเปอเรเตอร์ไว้ 2 โมดูลคือ โมดูลที่ทำหน้าที่กำหนดค่าโอเปอเรเตอร์กรณีอยู่ที่อินพุต (precedenceP) และ โมดูลที่ทำหน้าที่กำหนดค่าโอเปอเรเตอร์กรณีอยู่ในสแตก (precedenceST) แล้วนำค่าทั้งสองตำแหน่งมาเปรียบเทียบกับกันเพื่อดำเนินการตามขั้นตอนการแปลงดังระบุข้างต้น

2.3.1.1 โปรแกรมแปลงนิพจน์อินฟิกเป็นโอสฟิก

```
/*
Program convert infix to postfix by assigned in variable.
=====
*/
#include <stdio.h> //use printf()
#include <conio.h> //use getch()
#include <string.h> //use string function

#define MaxStack 40 //Set Max Operator Stack

char infix1[80] = {"A+B*(C^D*E/F)-G"}; //Assign INFIX
char OpSt[MaxStack]; //Operator stack size
int SP = 0; //Initial SP=0

void push(char oper) //PUSH Function
{
    if(SP == MaxStack) //Check Stack FULL?
    {
        printf("ERROR STACK OVER FLOW!!!...\n");
    }
    else
    {
        SP=SP+1; //Increase SP
        OpSt[SP]=oper; //Put data into Stack
    }
}
```

```

int pop() //POP Function
{
    char oper;
    if (SP != 0) //Check Stack NOT EMPTY?
    {
        oper=OpSt[SP]; //Get data from Stack
        SP--; //Decrease SP
        return(oper); //Return data
    }
    else
        printf("\nERROR STACK UNDER FLOW!!!...\n");
}

int precedenceIP(char oper) //Function for check precedence of input
operator
{
    switch (oper)
    {
        case '+' : return(1);
        case '-' : return(1);
        case '*' : return(2);
        case '/' : return(2);
        case '^' : return(4);
        case '(' : return(4);
    }
}

int precedenceST(char oper) //Function for check precedence of stack
operator
{
    switch (oper)
    {
        case '+' : return(1);
        case '-' : return(1);
        case '*' : return(2);
        case '/' : return(2);
        case '^' : return(3);
        case '(' : return(0);
    }
}

void infixTPostfix(char infix2[80])
{
    int i,j, len;
    char ch,temp;

```

```

printf("INFIX : %s\n ",infix2); //Show infix
len=strlen(infix2); //Find length of infix
printf("Infix Length = %d \n",len); //Display length of infix
printf("POSTFIX IS : ");
for (i=0;i<=len-1;i++) //split infix
{
    ch=infix2[i]; //Transfer character in to ch variable
    if (strchr("+-* /^()", ch)==0) //Check Is OPERAND?
        printf("%c",ch); //Out to Postfix
    else //If OPERATOR do below
    {
        if (SP==0) //Stack empty?
            push(ch); //Push any way if Stack empty
        else
            if (ch != ')') //If not ')' do below
            {
                if(precedenceIP(ch)>precedenceST(OpSt[SP])) //If
precedence input > precedence in stack
                    push(ch); //Push input operator to Stack
                else
                {
                    printf("%c",pop()); //Out to Postfix
                    while(precedenceIP(ch)<=precedenceST(OpSt[SP])
&& (SP != 0)) // Do Until precedence input > precedence in stack
                        printf("%c",pop()); //Out to Postfix
                    push(ch); //Push operator input to Stack
                }
            }
        else
        {
            temp=pop(); //Pop operator from Stack
            while ((temp != '(') ) // Do if not found '('
            {
                printf("%c",temp); //Out to Postfix
                temp=pop(); //Pop again and loop
            }
        }
    }
}
j=SP; //Let j for count left Stack
for(i=1;i<=j;i++) //POP all if Input is NULL
    printf("%c",pop()); //Out to Postfix
}

int main ()
{

```

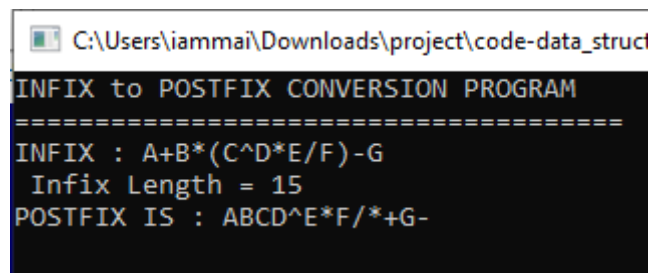


```

printf("INFIX to POSTFIX CONVERSION PROGRAM\n");
printf("=====\n");
infixTPostfix(infix1);
getch();
return(0);

} //End MAIN

```



```

C:\Users\iammai\Downloads\project\code-data_struct
INFIX to POSTFIX CONVERSION PROGRAM
=====
INFIX : A+B*(C^D*E/F)-G
Infix Length = 15
POSTFIX IS : ABCD^E*F/*+G-

```

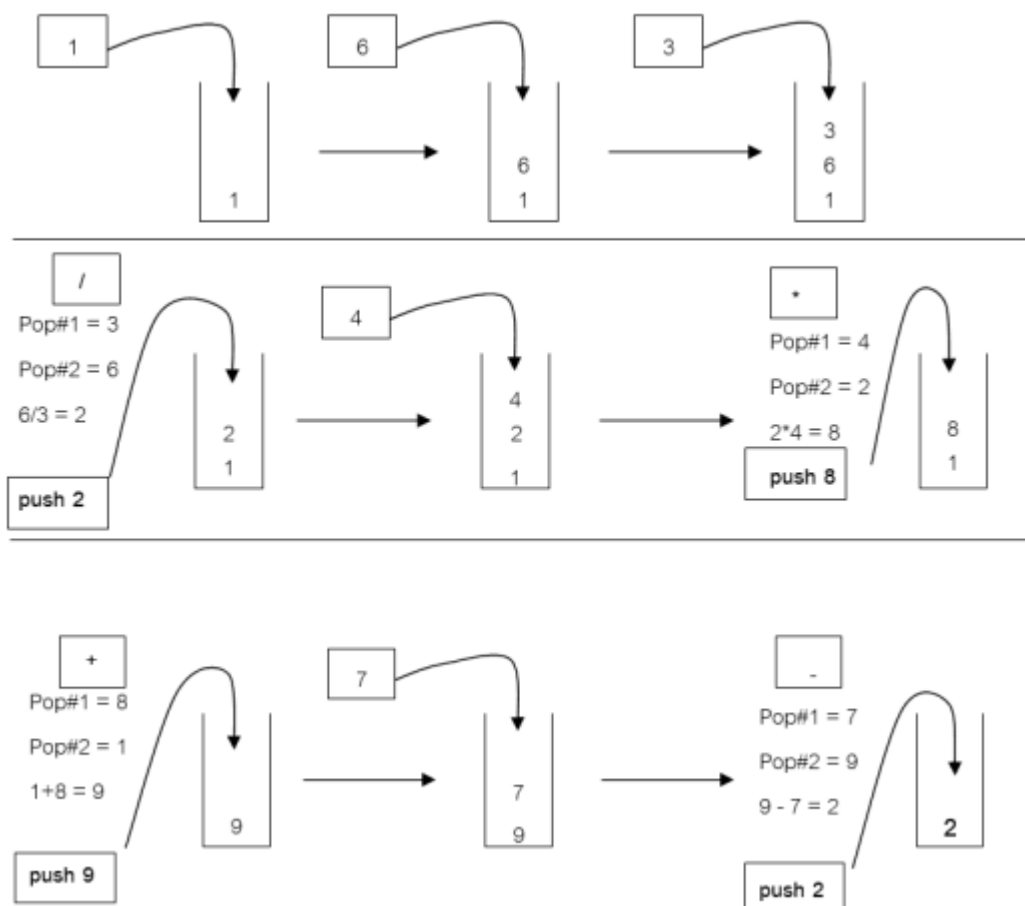
รูปที่ 2.28 แสดงผลการทำงานของโปรแกรมแปลงนิพจน์อินฟิกไปเป็นโพสฟิก

2.3.3 การหาค่าจากนิพจน์โพสฟิก

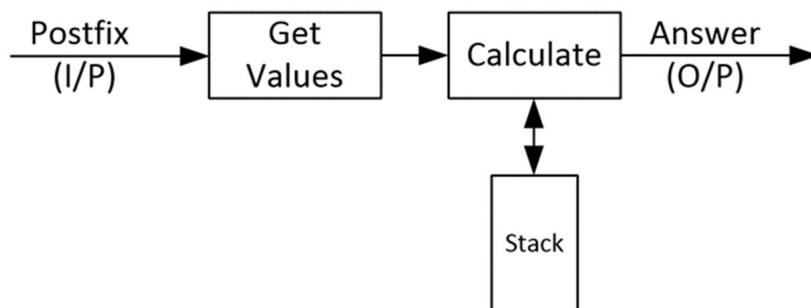
การหาค่าทางคณิตศาสตร์จากนิพจน์โพสฟิก มี 2 ขั้นตอน คือ 1. ถ้าอินพุตในโอเปอเรนด์ (Operand) ให้push (PUSH) สู่สแต็ก

2. ถ้าอินพุตเป็นโอเปอเรเตอร์ (Operator) ให้pop (POP) ค่า 2 ค่า จากสแต็ก โดยใช้การ pop ครั้งแรก (Pop#1) เป็นตัวกระทำ การ pop ครั้งที่สอง (Pop#2) เป็นตัวตั้ง แล้วคำนวณโดยโอเปอเรเตอร์ตัวนั้น และนำผลลัพธ์ที่ได้ push (Push) ลงสู่สแต็ก

ตัวอย่าง นิพจน์อินฟิก $A+B/C+D-E$ ถ้าแปลงเป็นนิพจน์โพสฟิกจะได้ $ABC/D*+E$ ถ้ากำหนดให้ $A=1, B=6, C=3, D=4$ และ $E=7$ จะมีนิพจน์โพสฟิก เป็น $1+6/3*4-7$ และ นิพจน์โพสฟิก เป็น $163/4*+7-$ สามารถหาค่าผลลัพธ์จากนิพจน์โพสฟิกได้ดังนี้



รูปที่ 2.29 แสดงขั้นตอนการหาค่าจากนิพจน์โพสฟิก



รูปที่ 2.30 ไตอะแกรมแสดงการหาค่านิพจน์โพสฟิก

การวางโครงสร้างโปรแกรมการหาค่าจากนิพจน์โพสฟิก

นิพจน์โพสฟิกจะกำหนดค่าไว้ในตัวแปรอาร์เรย์ชื่อ postfix1 ชนิด Character เพื่อความสะดวกในการแยกมาพิจารณาเป็นอินพุตทีละตัว และเนื่องจากนิพจน์โพสฟิกเป็น ตัวอักษร จึงต้องทำการ “ให้ค่า” กับตัวแปรแต่ละตัวก่อนที่จะนำไปคำนวณ โดยได้สร้างตัว แปรอาร์เรย์ 1 มิติ ชื่อ ValPostfix ชนิด Float อีก 1 ตัวเพื่อใช้เก็บค่าของตัวแปรของนิพจน์

โพสฟิก โดยให้ค่า Lower Bound กับ Upper Bound เหมือนกับตัวแปรที่ใช้เก็บนิพจน์โพสฟิก ดังรูป 2.26

	0	1	2	3	4	5	6	-----
Postfix (Text)	A	B	+	C	-	d	/	--
	0	1	2	3	4	5	6	-----
Value (Array)	20	30		10		8		--

รูปที่ 2.31 แสดงโครงสร้างข้อมูลที่ใช้เก็บนิพจน์และค่าของนิพจน์

ในขั้นตอนการให้ค่ากับนิพจน์โพสฟิกนี้ จะรับค่าเฉพาะที่เป็นโอเปอเรนด์เท่านั้น ส่วนโอเปอเรเตอร์จะไม่รับ ซึ่งจะใช้ฟังก์ชัน strchr ในการจำแนก จากนั้นก็นำค่าที่ได้มาทำการ Push และ Pop ในสแต็กเพื่อคำนวณหาค่าของนิพจน์ตามวิธีการดังกล่าวข้างต้น

2.3.3.1 โปรแกรมการหาค่าจากนิพจน์โพสฟิก

```

/*
Program calculate Postfix by assigned in variable.
=====
*/
#include <stdio.h> //use printf()
#include <conio.h> //use getch()
#include <string.h> //use string function
#include <math.h> //use power
#define MaxStack 40 //Set Max Operator Stack

char postfix1[80] ={"AB+C-d/"}; //Assign INFIX
float ValPostfix[80]; //Keep value of Postfix here
float ValOperandST[MaxStack]; //Operator stack size
int SP = 0; //Initial SP=0

```

```

void push(float ValOperand) //PUSH Function
{
    if(SP == MaxStack) //Check Stack FULL?
    {
        printf("ERROR STACK OVER FLOW!!!...\n");
    }
    else
    {
        SP=SP+1; //Increase SP
        ValOperandST[SP]=ValOperand; //Put data into Stack
    }
}

float pop() //POP Function
{
    float ValOperand;
    if (SP != 0) //Check Stack NOT EMPTY?
    {
        ValOperand=ValOperandST[SP]; //Get data from Stack
        SP--; //Decrease SP
        return(ValOperand); //Return data
    }
    else
        printf("\nERROR STACK UNDER FLOW!!!...\n");
}

void CalPostfix(char postfix[80])
{
    float pop1,pop2,value;
    int i,len;
    char ch;
    len = strlen(postfix);
    printf("Postfix = %s\n",postfix);
    for (i=0;i<=len-1;i++) //Assign data to OPERAND
    {
        ch=postfix[i]; //Split Character for assign data
        if (strchr("+-* /^", ch)==0) //Check Is OPERAND?
        {
            printf("\nAssign Number to %c : ",ch);
            scanf("%f",&ValPostfix[i]); //Read data from KBD and direct
to Value of OPERAND in Array
        }
    }
    for (i=0;i<=len-1;i++) //Calculate Value of POSTFIX
    {
        ch=postfix[i]; //Split Character for prepare to STACK

```

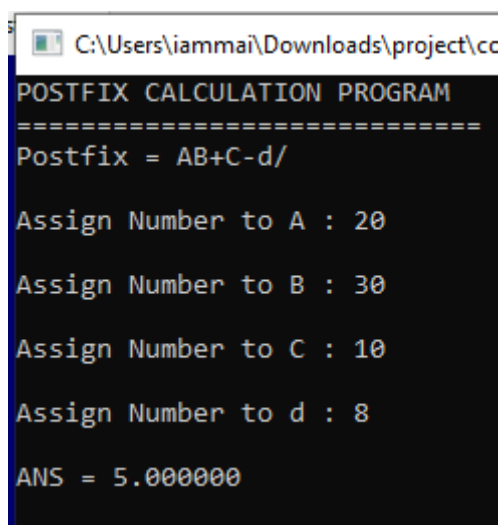
```

if (strchr("+-* /^", ch)==0) //Check Is OPERAND?
{
    push(ValPostfix[i]); //push value of OPERAND to Stack
}
else
{
    pop1=pop(); //Pop 1st
    pop2=pop(); //pop 2nd
    switch (ch)
    {
        case '+': value=pop2+pop1; //Calculate
                  push(value); //Push value to Stack
                  break;
        case '-': value=pop2-pop1;
                  push(value);
                  break;
        case '*': value=pop2*pop1;
                  push(value);
                  break;
        case '/': value=pop2/pop1;
                  push(value);
                  break;
        case '^': value=pow(pop2,pop1);
                  push(value);
                  break;
    }
    } //End IF
} //End IF
printf("\nANS = %f",pop()); //Last value is ANSWER
} //End Fn.

int main()
{
    printf("POSTFIX CALCULATION PROGRAM\n");
    printf("=====\n");
    CalPostfix(postfix1);
    getch();
    return(0);

} //End Main

```



```

C:\Users\iammai\Downloads\project\cc
POSTFIX CALCULATION PROGRAM
=====
Postfix = AB+C-d/

Assign Number to A : 20
Assign Number to B : 30
Assign Number to C : 10
Assign Number to d : 8
ANS = 5.000000

```

รูปที่ 2.32 แสดงผลการทำงานของโปรแกรมหาค่าจากนิพจน์โพสฟิก

2.4 การเขียนโปรแกรมเรียกตนเอง (Recursive Programming)

ปกติแล้วการเขียนโปรแกรมที่มีส่วนที่ต้องทำซ้ำซ้อนสามารถทำได้โดยใช้ หลักการทำซ้ำซ้อน ด้วยคำสั่ง DO LOOP หรือ DO WHILE หรือคำสั่งอื่น ๆ ในลักษณะ เดียวกัน การเขียนโปรแกรมแบบนี้ว่า โปรแกรมวนซ้ำ (iterative Program) บางครั้งการ เขียนโปรแกรมแก้ปัญหาจะ สะดวกมากกว่าเมื่อเขียนโปรแกรมแบบรีเคอร์ซีฟ (Recursive) หรือ โปรแกรมเรียกตนเอง

งานที่เหมาะสมในการเขียนโปรแกรมแบบเรียกตนเอง

รีเคอร์ซีฟเป็นวิธีการที่สำคัญอย่างหนึ่งของการเขียนโปรแกรม ซึ่งอัลกอริธึมหลาย อย่างสามารถใช้วิธีการของรีเคอร์ซีฟอธิบายได้ โดยรีเคอร์ซีฟมีคุณสมบัติดังนี้

1. จะต้องมีการกำหนดบางอย่าง ซึ่งเรียกว่า Base Criteria สำหรับหยุดการวนเรียกตนเอง
2. ในแต่ละครั้งที่เรียกตนเอง ค่าที่ได้จะต้องเข้าใกล้กับกฎเกณฑ์ของ Base criteria

2.4.1 การวิเคราะห์งานฟังก์ชันแฟกทอเรียล (Factorial Function)

การหาค่า $N!$ คือ ผลคูณของเลขจำนวนเต็มจาก N ถึง 1 ดังนี้

เช่น $N = 5$ คือ $5! = 5 * 4 + 3 + 2 + 1 = 120$

หรือ

$N! = N(N-1)(N-2) \dots 1$ หรือ $N! = N(N-1)! และ 0!=1$

ซึ่งเป็นค่า Base Criteria

ขั้นตอนการวิเคราะห์การเขียนโปรแกรมเรียกตนเองฟังก์ชันแฟกทอเรียล

(a) If $N = 0$ then $N! = 1$

(b) If $N > 0$ then $N! = N * (N-1)!$

สังเกตว่านิยามนี้ $N!$ เป็นรีเคอร์ซีฟ เนื่องจากมีการเรียกตัวเองเมื่อใช้ $(N-1)$ แต่ อย่างไรก็ตามในบรรทัด (a) จะให้ค่า $N!$ ออกมาเมื่อ $N = 0$ (เนื่องจาก 0 เป็น Base Value) และค่าของ $N!$ ในแต่ละค่าของ N ใด ๆ ในบรรทัด (b) จะอยู่ในเทอมของ N ที่น้อยกว่า แต่ไม่เท่ากับ 0 ถ้าเท่ากับ 0 ก็จะทำให้เป็นไปตามนิยาม (a) ซึ่งจะเป็นเงื่อนไขของการ หยุดเรียกตนเอง โดยสามารถนำมาผู้เป็นโครงสร้างในการส่งงานดังนี้

FACT : Procedure(N) Recursive returns(Fixed)

/* ให้ _ N , X, Y เป็พวนจำนวนเต็ม*/

IF $N=0$ THEN RETURN(1)

$X = N - 1$

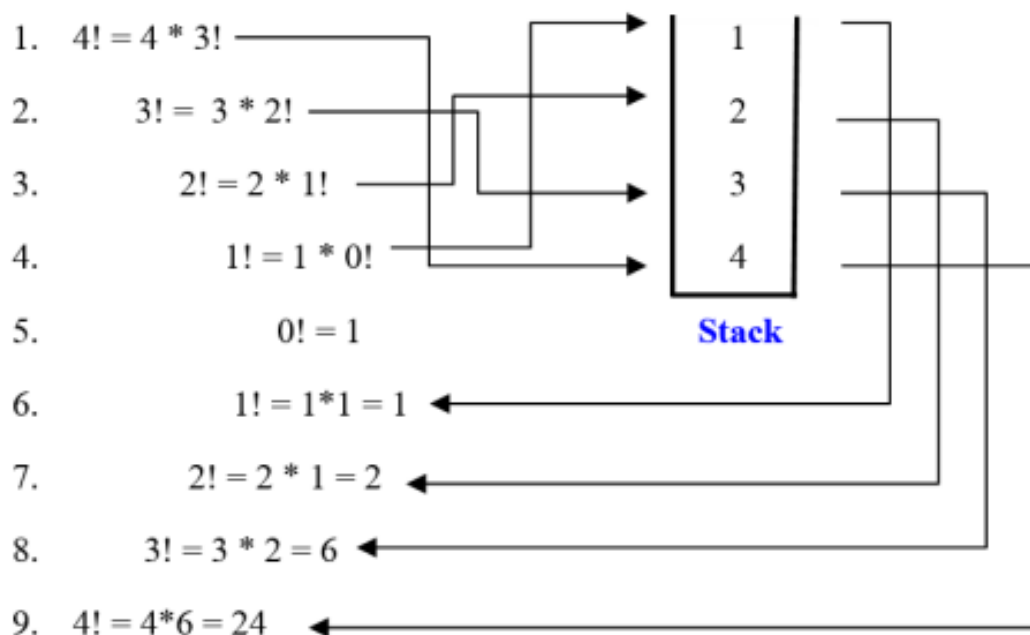
$Y = \text{FACT}(X)$

RETURN ($N * Y$)

ENDIF

END : FACT

ตัวอย่างการคำนวณหาค่า $4!$ โดยใช้นิยามรีเคอร์ซีฟคำนวณใช้ 9 ขั้นตอนดังนี้



ขั้นตอนการคำนวณ

ขั้นตอนที่ 1 : กำหนด 4 ในเทอมของ 3! ซึ่งยังไม่หาค่าของ 4! จนกว่าจะทราบค่าของ 3!

ขั้นตอนที่ 2 : ค่า 3 กำหนดในเทอมของ 2! ซึ่งยังไม่ทราบค่า 3! จนกว่าจะทราบค่า

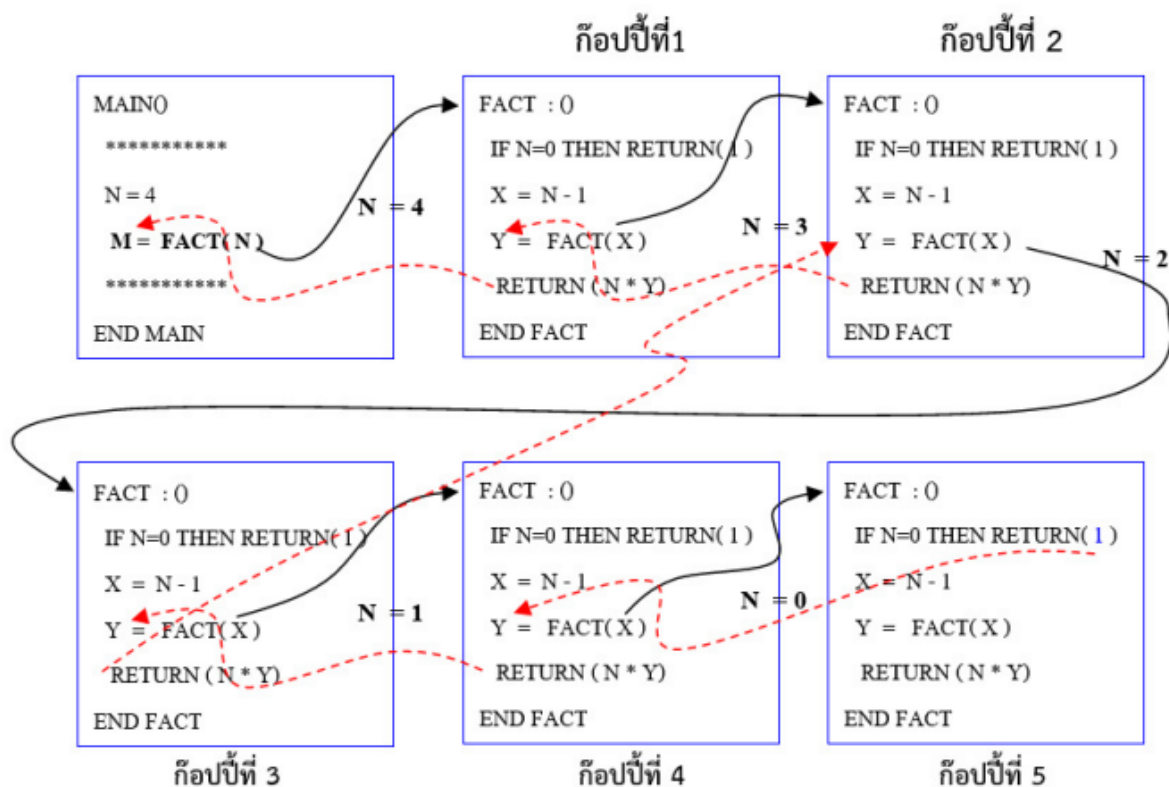
ขั้นตอนที่ 3 : กำหนด 2 ในเทอมของ 1! ซึ่งยังไม่ทราบค่า 2! จนกว่าจะทราบค่า 1!

ขั้นตอนที่ 4 : กำหนด 1 ในเทอมของ 0! ซึ่งยังไม่ทราบค่า 1! จนกว่าจะทราบค่า 0!

ขั้นตอนที่ 5 : ให้ค่า $0! = 1$ เนื่องจาก 0 เป็น Base Value ของนิยาม

ขั้นตอนที่ 6-9 : เป็นการทำกลับ โดยใช้ 0! ในการหาค่า 1! และใช้ค่า 1! ในการหาค่า 2! ใช้ค่า 2! ที่ได้ไปหาค่า 3! และใช้ค่า 3! ไปหาค่า 4! ตามลำดับ

ค่า N แต่ละรอบที่ทราบค่าจะถูกนำไปเก็บไว้ในสแต็กก่อนจนกว่าจะทราบค่า Base Value จากนั้นจึงอ่านข้อมูลจากสแต็กเพื่อนำมาคำนวณหาค่าย้อนกลับ สุดท้ายก็จะได้คำตอบ ซึ่งกระบวนการนี้คอมพิวเตอร์จะเป็นผู้ดำเนินการทั้งสิ้น (หมายเหตุ -บางภาษาก็ไม่รองรับการเขียนโปรแกรมแบบเรียกตนเอง)



รูปที่ 2.33 แสดงการเรียกใช้โปรแกรมเรียกตนเอง

2.4.1.1 โปรแกรมหาค่าแฟกตอเรียล

```

/*
Program Recursion can..
1. Calculate Factorial Function
2. Show each step of recursive result
3. Show final result
=====
=====
*/
#include <stdio.h> //use printf()
#include <conio.h> //use getch()

int Number , ans ;

int Factorial( int N ) { //Factorial Function
    int x , y ;
    if( N == 0 ) {
        printf( ".....Roll Back Point\n" ) ;
        return( 1 ) ;
    } else {
        x = N - 1 ;
        printf( "%2d! = %2d * %2d!\n" , N , N , x ) ; //Display
before Recursive
        y = Factorial( x ) ;
        printf( "%2d! = %2d * %3d = %5d\n" , N , N , y , y * N )
; //Display After Recursive
        return( N * y ) ;
    }
} //End Fn

int main() {
    printf( "RECRSIVE(FACTORIAL) PROGRAM\n" ) ;
    printf( "=====\n" ) ;
    //N=0;
    while( Number != -999 ) {
        printf( "Enter Number (-999 is END) : " ) ;
        scanf( "%d" , &Number ) ;
        if( Number >= 0 ) {
            printf( "N! = N(N-1)!\n" ) ;
            printf( "-----\n" ) ;
            ans = Factorial( Number ) ; //Recursive it self
            printf( "\nAnswer N! = %d\n" , ans ) ;
            printf( "-----Finished\n" ) ;
            getch() ;
        }
    }
}

```

```

    } //END while
    return(0);
} //End Main

```

```

C:\Users\iammai\Downloads\project\code-data_structure\
RECRSIVE(FACTORIAL) PROGRAM
=====
Enter Number (-999 is END) : 6
N! = N(N-1)!
-----
6! = 6 * 5!
5! = 5 * 4!
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!
.....Roll Back Point
1! = 1 * 1 = 1
2! = 2 * 1 = 2
3! = 3 * 2 = 6
4! = 4 * 6 = 24
5! = 5 * 24 = 120
6! = 6 * 120 = 720

Answer N! = 720
-----Finished

```

รูปที่ 2.34 แสดงผลการทำงานของโปรแกรมเรียกตนเอง

แบบฝึกหัดที่ 2

1. จงอธิบายความหมายและคุณสมบัติของสแตก (Stack) พร้อมยกตัวอย่าง ?
2. จงอธิบายวิธีการแทรก (Insert) และการลบ (Delete) ของคิวธรรมดาพร้อมวาดรูปประกอบ ?
3. จงอธิบายความหมายและคุณสมบัติของคิววงกลมพร้อมยกตัวอย่าง ?
4. จงแสดงวิธีการแปลง $(A + ((B * C * D / E) - F) * G) / H^2$ ไปเป็นนิพจน์โพสฟิก และ แสดงวิธีการหาผลลัพธ์ของนิพจน์โพสฟิกที่หามาได้โดยแทนค่า $A=5, B=2, C=3, D=5, E=3, F=1, G=5, H=5$ มาตามลำดับ ?
5. ทำไมนิพจน์โพสฟิก จึง “ไม่มีเครื่องหมายวงเล็บ” จงอธิบายพร้อมให้เหตุผล ประกอบ ?
6. ทำไมจึงใช้โครงสร้างสแตกในการเก็บโอเปอเรเตอร์ (Operator) ในการแปลงนิพจน์ อินฟิก ไปเป็นโพสฟิก ?
7. ค่า Base Criteria คืออะไรในงานเขียนโปรแกรมแบบวนเรียกตนเอง (Recursive)
8. ทำไมการเขียนโปรแกรมแบบวนเรียกตนเอง (Recursive) จึงใช้ได้เฉพาะงานที่ทราบ ค่า Base Criteria เท่านั้น ?
9. จงอธิบายหลักการของการเขียนโปรแกรมแบบวนเรียกตนเอง (Recursive) พร้อมยกตัวอย่างการทำงานของโปรแกรมวนเรียกตนเองมาเป็นขั้นตอน ?
10. จงเขียนอัลกอริธึมการเรียกตนเอง ของ M^N ใด ๆ พร้อมทั้งอธิบายขั้นตอนการทำงานมาตามลำดับโดยละเอียด ?
11. ทำไมจึงใช้โครงสร้างสแตกในการทำงานของโปรแกรมแบบวนเรียกตนเอง (Recursive) และใช้เมื่อใดที่ไหน ?