// Stack Program A

```
#include<stdio.h>
#include<stdlib.h>
#define Size 4

int Top=-1, inp_array[Size];
void Push();
void Pop();
void show();

int main()
{
   int choice;
   while(1)
   {
          printf("\nOperations performed by Stack");
          printf("\n1.Push the element\n2.Pop the element\n3.Show\n4.End");
          printf("\n\nEnter the choice:");
          scanf("%d",&choice);
          switch(choice)
          {
                 case 1: Push();
                        break;
                 case 2: Pop();
                        break;
                 case 3: show();
                        break;
                 case 4: exit(0);

                 default: printf("\nInvalid choice!!");
          }
   }
}

void Push()
{
   int x;

   if(Top==Size-1)
   {
          printf("\nOverflow!!");
   }
   else
   {
```

```
        printf("\nEnter element to be inserted to the stack:");
        scanf("%d",&x);
        Top=Top+1;
        inp_array[Top]=x;
    }
}

void Pop()
{
    if(Top==-1)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nPopped element:  %d",inp_array[Top]);
        Top=Top-1;
    }
}

void show()
{
    if(Top==-1)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nElements present in the stack: \n");
        for(int i=Top;i>=0;--i)
            printf("%d\n",inp_array[i]);
    }
}
```

// Stack Program B

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10

int count = 0;

// Creating a stack
```

```
struct stack {
  int items[MAX];
  int top;
};
typedef struct stack st;

void createEmptyStack(st *s) {
  s->top = -1;
}

// Check if the stack is full
int isfull(st *s) {
  if (s->top == MAX - 1)
          return 1;
  else
          return 0;
}

// Check if the stack is empty
int isempty(st *s) {
  if (s->top == -1)
          return 1;
  else
          return 0;
}

// Add elements into stack
void push(st *s, int newitem) {
  if (isfull(s)) {
          printf("STACK FULL");
  } else {
          s->top++;
          s->items[s->top] = newitem;
  }
  count++;
}

// Remove element from stack
void pop(st *s) {
  if (isempty(s)) {
          printf("\n STACK EMPTY \n");
  } else {
          printf("Item popped= %d", s->items[s->top]);
          s->top--;
```

```
  }
  count--;
  printf("\n");
}

// Print elements of stack
void printStack(st *s) {
  printf("Stack: ");
  for (int i = 0; i < count; i++) {
          printf("%d ", s->items[i]);
  }
  printf("\n");
}

// Driver code
int main() {
  int ch;
  st *s = (st *)malloc(sizeof(st));

  createEmptyStack(s);

  push(s, 1);
  push(s, 2);
  push(s, 3);
  push(s, 4);

  printStack(s);

  pop(s);

  printf("\nAfter popping out\n");
  printStack(s);
}


#include<stdio.h>
#include<stdlib.h>

struct stack{
          int size ;
          int top;
          int * arr;
};
```

```
int isEmpty(struct stack* ptr){
	if(ptr->top == -1){
	return 1;
	}
	else{
	return 0;
	}
}

int isFull(struct stack* ptr){
	if(ptr->top == ptr->size - 1){
	return 1;
	}
	else{
	return 0;
	}
}

void push(struct stack* ptr, int val){
	if(isFull(ptr)){
	printf("Stack Overflow! Cannot push %d to the stack\n", val);
	}
	else{
	ptr->top++;
	ptr->arr[ptr->top] = val;
	}
}

int pop(struct stack* ptr){
	if(isEmpty(ptr)){
	printf("Stack Underflow! Cannot pop from the stack\n");
	return -1;
	}
	else{
	int val = ptr->arr[ptr->top];
	ptr->top--;
	return val;
	}
}

int main(){
	struct stack *sp = (struct stack *) malloc(sizeof(struct stack));
	sp->size = 6;
	sp->top = -1;
```

```
    sp->arr = (int *) malloc(sp->size * sizeof(int));
    printf("Stack has been created successfully\n");
    printf("The stack is full %d",isFull(sp));
    printf("The stack is empty %d",isEmpty(sp));
    push(sp, 3);
    push(sp, 2);
    push(sp, 9);
    push(sp, 5);
    push(sp, 17);
    push(sp, 6);
    push(sp, 57);
    push(sp, 6);
    push(sp, 29);
    printf("Popped %d from the stack\n", pop(sp));
    printf("Popped %d from the stack\n", pop(sp));
    printf("Popped %d from the stack\n", pop(sp));
    printf("Popped %d from the stack\n", pop(sp));
    printf("Popped %d from the stack\n", pop(sp));
    printf("Popped %d from the stack\n", pop(sp));
    printf("Popped %d from the stack\n", pop(sp));
    printf("Popped %d from the stack\n", pop(sp));

    return 0;
}
```

Conversion of infix to postfix

Here, we will use the stack data structure for the conversion of infix expression to prefix expression. Whenever an operator will encounter, we push the operator into the stack. If we encounter an operand, then we append the operand to the expression.

Rules for the conversion from infix to postfix expression

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3. If the incoming symbol is '(', push it on to the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.

7. If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
8. At the end of the expression, pop and print all the operators of the stack.

Let's understand through an example.

Infix expression: K + L - M*N + (O^P) * W/U/V * T + Q

Input Expression	Stack	Postfix Expression
K		K
+	+	
L	+	K L
-	-	K L+
M	-	K L+ M
*	- *	K L+ M
N	- *	K L + M N
+	+	K L + M N* K L + M N* -
(	+ (	K L + M N *-
O	+ (	K L + M N * - O
^	+ (^	K L + M N* - O
P	+ (^	K L + M N* - O P
)	+	K L + M N* - O P ^
*	+ *	K L + M N* - O P ^
W	+ *	K L + M N* - O P ^ W
/	+ /	K L + M N* - O P ^ W *
U	+ /	K L + M N* - O P ^W*U
/	+ /	K L + M N* - O P ^W*U/

V	+ /	KL + MN*-OP^W*U/V
*	+ *	KL+MN*-OP^W*U/V/
T	+ *	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*Q
		KL+MN*-OP^W*U/V/T*+Q+

The final postfix expression of infix expression(K + L - M*N + (O^P) * W/U/V * T + Q) is KL+MN*-OP^W*U/V/T*+Q+.

What is infix notation?

An infix notation is a notation in which an expression is written in a usual or normal format. It is a notation in which the operators lie between the operands. The examples of infix notation are A+B, A*B, A/B, etc.

As we can see in the above examples, all the operators exist between the operands, so they are infix notations. Therefore, the syntax of infix notation can be written as:

<operand> <operator> <operand>

Parsing Infix expressions

In order to parse any expression, we need to take care of two things, i.e., **Operator precedence** and **Associativity**. Operator precedence means the precedence of any operator over another operator. For example:

A + B * C → A + (B * C)

As the multiplication operator has a higher precedence over the addition operator so B * C expression will be evaluated first. The result of the multiplication of B * C is added to the A.

Precedence order

Operators	Symbols
Parenthesis	{ }, ( ), [ ]

Exponential notation	^
Multiplication and Division	*, /
Addition and Subtraction	+, -

Associativity means when the operators with the same precedence exist in the expression. For example, in the expression, i.e., A + B - C, '+' and '-' operators are having the same precedence, so they are evaluated with the help of associativity. Since both '+' and '-' are left-associative, they would be evaluated as (A + B) - C.

Associativity order

Operators	Associativity
^	Right to Left
*, /	Left to Right
+, -	Left to Right

Let's understand the associativity through an example.

1 + 2*3 + 30/5

Since in the above expression, * and / have the same precedence, so we will apply the associativity rule. As we can observe in the above table that * and / operators have the left to right associativity, so we will scan from the leftmost operator. The operator that comes first will be evaluated first. The operator * appears before the / operator, and multiplication would be done first.

1+ (2*3) + (30/5)

1+6+6 = 13

What is Prefix notation?

A prefix notation is another form of expression but it does not require other information such as precedence and associativity, whereas an infix notation requires information of precedence and associativity. It is also known as **polish notation**. In prefix notation, an operator comes before the operands. The syntax of prefix notation is given below:

<operator> <operand> <operand>

For example, if the infix expression is 5+1, then the prefix expression corresponding to this infix expression is +51.

If the infix expression is:

a * b + c

↓

***ab+c**

↓

+*abc (Prefix expression)

Consider another example:

A + B * C

First scan: In the above expression, multiplication operator has a higher precedence than the addition operator; the prefix notation of B*C would be (*BC).

A + *BC

Second scan: In the second scan, the prefix would be:

+A *BC

In the above expression, we use two scans to convert infix to prefix expression. If the expression is complex, then we require a greater number of scans. We need to use that method that requires only one scan, and provides the desired result. If we achieve the desired output through one scan, then the algorithm would be efficient. This is possible only by using a stack.

Conversion of Infix to Prefix using Stack

K + L - M * N + (O^P) * W/U/V * T + Q

If we are converting the expression from infix to prefix, we need first to reverse the expression.

The Reverse expression would be:

Q + T * V/U/W *) P^O(+ N*M - L + K

To obtain the prefix expression, we have created a table that consists of three columns, i.e., input expression, stack, and prefix expression. When we encounter any symbol, we simply add it into the prefix expression. If we encounter the operator, we will push it into the stack.

Input expression	Stack	Prefix expression
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVUW
P	+*//*)	QTVUWP
^	+*//*)^	QTVUWP
O	+*//*)^	QTVUWPO
(	+*//*	QTVUWPO^
+	++	QTVUWPO^*//*
N	++	QTVUWPO^*//*N
*	++*	QTVUWPO^*//*N
M	++*	QTVUWPO^*//*NM
-	++-	QTVUWPO^*//*NM*
L	++-	QTVUWPO^*//*NM*L
+	++-+	QTVUWPO^*//*NM*L
K	++-+	QTVUWPO^*//*NM*LK

QTVUWPO^*//*NM*LK+-++

The above expression, i.e., QTVUWPO^*//*NM*LK+-++, is not a final expression. We need to reverse this expression to obtain the prefix expression.

Rules for the conversion of infix to prefix expression:

- First, reverse the infix expression given in the problem.
- Scan the expression from left to right.
- Whenever the operands arrive, print them.
- If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
- If the incoming operator has higher precedence than the TOP of the stack, push the incoming operator into the stack.
- If the incoming operator has the same precedence with a TOP of the stack, push the incoming operator into the stack.
- If the incoming operator has lower precedence than the TOP of the stack, pop, and print the top of the stack. Test the incoming operator against the top of the stack again and pop the operator from the stack till it finds the operator of a lower precedence or same precedence.
- If the incoming operator has the same precedence with the top of the stack and the incoming operator is ^, then pop the top of the stack till the condition is true. If the condition is not true, push the ^ operator.
- When we reach the end of the expression, pop, and print all the operators from the top of the stack.
- If the operator is ')', then push it into the stack.
- If the operator is '(', then pop all the operators from the stack till it finds) opening bracket in the stack.
- If the top of the stack is ')', push the operator on the stack.
- At the end, reverse the output.

Pseudocode of infix to prefix conversion

1. Function InfixtoPrefix(stack, infix)
2. infix = reverse(infix)
3. loop i = 0 to infix.length
4. if infix[i] is operand → prefix+= infix[i]
5. else if infix[i] is '(' → stack.push(infix[i])
6. else if infix[i] is ')' → pop and print the values of stack till the symbol ')' is not found
7. else if infix[i] is an operator(+, -, *, /, ^) →
8.
9. if the stack is empty then push infix[i] on the top of the stack.

10. Else →
11. If precedence(infix[i] > precedence(stack.top))
12. → Push infix[i] on the top of the stack
13. else if(infix[i] == precedence(stack.top) && infix[i] == '^')
14. → Pop and print the top values of the stack till the condition is true
15. → Push infix[i] into the stack
16. else if(infix[i] == precedence(stack.top))
17. → Push infix[i] on to the stack
18. Else if(infix[i] < precedence(stack.top))
19. → Pop the stack values and print them till the stack is not empty and infix[i] < precedence(stack.top)
20. → Push infix[i] on to the stack
21. End loop
22. Pop and print the remaining elements of the stack
23. Prefix = reverse(prefix)
24. return