

Prefix – Infix – Postfix

Prefix: +ab

Infix: a+b

Postfix: ab+

Q: Convert the following infix expression into their corresponding postfix form:

1> $a + b * (c * d - e) / f - g$
2> $a + b + c * d / e$
3> $a + (b * c - d * e) / f - g / h$
4> $(a * b - c * d) / (e - f) + g$
5> $a + b * c * d - (e - f / g) * h$

Solution:

1> $a + b * (c * d - e) / f - g$
a + b * ([cd*] - e) / f - g
a + b * [cd*e-] / f - g
a + [bcd*e-*] / f - g
a + [bcd*e-*f/] - g
[abcd*e-*f/+] - g
a b c d * e - * f / + g -

Q: Convert the following infix expression into their corresponding prefix form:

1> $a + b * (c * d - e) / f - g$
2> $a + b + c * d / e$
3> $a + (b * c - d * e) / f - g / h$
4> $(a * b - c * d) / (e - f) + g$
5> $a + b * c * d - (e - f / g) * h$

Solution:

1> $a + b * (c * d - e) / f - g$
a + b * ([*cd] - e) / f - g
a + b * [-*cde] / f - g
a + [*b-*cde] / f - g
a + [/*b-*cdef] - g
[+a/*b-*cdef] - g
- + a / * b - * c d e f g

1: Write an algorithm to evaluate postfix or reverse polish expression.

STEP 1: Insert # at the end of the postfix expression

STEP 2: Scan the postfix expression from left to right and for each character repeat STEP 3 and STEP 4 until # is encountered.

STEP 3: If an operand is encountered, push it onto STACK.

STEP 4: If an operator Θ is encountered, pop the top element **A** and next to top Element **B** and perform the following operations:

- A. Evaluate $X = B \Theta A$
- B. Push **X** onto STACK

STEP 5: Top element of the stack gives the result of the postfix expression.

STEP 6: Exit.

/* WRITE A PROGRAM TO EVALUATE POSTFIX EXPRESSION OR REVERSE POLISH EXPRESSION. */

```
# include <stdio.h>
# include <math.h>
# include <stdlib.h>
# define MAX 100

struct stack {
    int info[MAX];
    int top;
};
typedef struct stack STK;

int evaluate_postfix(char[]);
int eval(int ,int, char);
int is_digit(char);
void push(STK *,int);
int pop(STK *);

int main(void) {
    char postfix[MAX];
    int x;

    printf("\n Give the postfix expression: ");
    scanf("%s",postfix);
    x=evaluate_postfix(postfix);
    printf("\n Result of the postfix expression %s is %d",postfix ,x);

    return 0;
}

int evaluate_postfix(char postfix[]) {
    STK stk;
    int i;

    stk.top=-1;
    i=0;
    while(postfix[i]!='\0') {
        if(is_digit(postfix[i])) {
            push( &stk, postfix[i]-'0');
        }
        else {
            push( &stk, eval(pop(&stk),pop(&stk),postfix[i]));
        }
        i++;
    }
```

```
    }
    return pop(&stk);
}

int eval(int b,int a,char op) {
    switch(op) {
        case '+':return (b+a);
        case '-':return (b-a);
        case '*':return (b*a);
        case '/':return (b/a);
        case '$':return (pow(b,a));
        default:printf("\n Invalid operator %c",op);
                exit(1);
    }
}

int is_digit(char x) {
    return (x>='0'&&x<='9');
}

void push(STK *p,int item) {
    if(p->top==MAX-1) {
        printf("\n Stack overflow");
        exit(1);
    }
    p->info[++p->top]=item;
}

int pop(STK *p) {
    if(p->top== -1) {
        printf("\n Stack underflow");
        exit(1);
    }
    return (p->info[p->top--]);
}
```

EXAMPLE:

Infix: $3 + 4 * 2 \$ 3 - (5 - 6 / 2) * 6$

Postfix: $3423\$*+562/-6*-$

Result: 23

2: Write an algorithm to evaluate prefix or polish expression.

STEP 1: Insert # at the beginning of the prefix expression

STEP 2: Scan the prefix expression from right to left and for each character repeat STEP 3 and STEP 4 until # is encountered.

STEP 3: If an operand is encountered, push it onto STACK.

STEP 4: If an operator Θ is encountered, pop the top element **A** and next to top Element **B** and perform the following operations:

A. Evaluate $X = A \ominus B$

B. Push X onto STACK

STEP 5: Top element of the stack gives the result of the prefix expression.

STEP 6: Exit.

/* WRITE A PRTOGRAM TO EVALUATE THE PREFIX EXPRESSION. */

```
# include <stdio.h>
# include <string.h>
# include <math.h>
# include <stdlib.h>
# define MAX 20

struct stack {
    int info[MAX];
    int top;
};
typedef struct stack STK;

int evaluate_prefix(char[]);
int eval(int ,int, char);
int is_digit(char);
void push(STK *,int);
int pop(STK *);

int main(void) {
    char prefix[MAX];
    int x;

    printf("\n Give the prefix expression: ");
    scanf("%s",prefix);
    x=evaluate_prefix(prefix);
    printf("\n Result of the prefix expression %s is %d",prefix ,x);

    return 0;
}

int evaluate_prefix(char prefix[]) {
    STK stk;
    int i;
    stk.top=-1;
    i=strlen(prefix)-1;
    while(i>=0) {
        if (is_digit(prefix[i])) {
            push(&stk,prefix[i]-'0');
        } else {
            push(&stk,eval(pop(&stk),pop(&stk),prefix[i]));
        }
        i--;
    }
```

```
    }  
    return pop(&stk);  
}  
  
int eval(int b,int a,char op) {  
    switch(op) {  
        case '+':return (a+b);  
        case '-':return (a-b);  
        case '*':return (a*b);  
        case '/':return (a/b);  
        case '$':return (pow(a,b));  
        default:printf("\n Invalid operator %c",op);  
                exit(1);  
    }  
}  
  
int is_digit(char x) {  
    return (x>='0'&&x<='9');  
}  
  
void push(STK *p,int item) {  
    if(p->top==MAX-1) {  
        printf("\n Stack overflow");  
        exit(1);  
    }  
    p->info[++p->top]=item;  
}  
  
int pop(STK *p) {  
    if(p->top==-1) {  
        printf("\n Stack underflow");  
        exit(1);  
    }  
    return (p->info[p->top--]);  
}
```

EXAMPLE:

Infix: $3 + 4 * 2 \$ 3 - (5 - 6 / 2) * 6$

Prefix: $-+3*4\$23*-5/626$

Result: 23

3: Write an algorithm to convert an infix expression into postfix or reverse polish expression.

STEP 1: Insert a right parenthesis ')' at the end of the infix expression and push a left parenthesis '(' onto STACK.

STEP 2: Scan the infix expression from left to right and for each character repeat STEP 3 to STEP 6 until the STACK is empty.

STEP 3: If an operand is encountered, write it into the output string.

STEP 4: If a left parenthesis '(' is encountered, push it onto the STACK.

STEP 5: If an operator is encountered, repeatedly pop operators (if any) from STACK whose precedence are greater than or equal to that of encountered operator. Write those popped operators (if any) into output string. Then push the encountered operator onto STACK.

STEP 6: If a right parenthesis ')' is encountered, repeatedly pop operators (if any) from stack until a left parenthesis '(' is popped from the STACK. Write those popped operators (if any) into output string.

STEP 7: The output string gives the required postfix expression.

STEP 8: Exit.

/* WRITE A PROGRAM TO CONVERT INFIX EXPRESSION INTO POSTFIX (REVERSE POLISH) EXPRESSION. */

```
# include <stdio.h>
# include <string.h>
# include <stdlib.h>
# define MAX 20

struct stack {
    char info[MAX];
    int top;
};
typedef struct stack STK;

void infix_to_postfix(char[], char[]);
int isoperator(char);
int precedence(char);
void push(STK *, char);
char pop(STK *);

int main(void) {
    char infix[MAX], postfix[MAX];
    printf("\nGive an infix expression: ");
    scanf("%s", infix);
    infix_to_postfix(infix, postfix);
    printf("\nPostfix exp. of %s is %s", infix, postfix);
    return 0;
}

void infix_to_postfix(char a[], char b[]) {
    STK stk;
    int i, j, l;
    char x;

    stk.top = -1;
    l = strlen(a);
    a[l] = '\0';
    push(&stk, '(');
    i = j = 0;
```

```
while(stk.top != -1) {
    printf("\nScanning character %c", a[i]);
    if( a[i] == '(' ) {
        push( &stk, a[i++] );
    } else {
        if( isoperator( a[i] ) ) {
            while(precedence(stk.info[stk.top]) >= precedence(a[i])) {
                b[j++] = pop( &stk );
            }
            push( &stk, a[i]);
        } else {
            if(a[i] == ')') {
                while( ( x=pop( &stk ) ) != '(' ) {
                    b[j++] = x;
                }
            } else {
                b[j++] = a[i];
            }
        }
        i++;
    }
}
b[j] = '\0';
a[l] = '\0';
}
```

```
int isoperator(char x) {
    return ( x=='+' || x=='-' || x=='*' || x=='/' || x=='$' );
}
```

```
int precedence(char op) {
    switch( op ) {
        case '+':
        case '-':
            return (1);
        case '*':
        case '/':
            return (2);
        case '$':
            return (3);
        case '(':
            return (0);
        default:
            printf("\n Invalid operator %c", op );
            exit(1);
    }
}
```

```
void push(STK *p, char item) {
    printf("\nItem pushed %c", item);
    if( p->top == MAX-1 ) {
```

```
        printf("\n Stack Overflow");
        return;
    }
    p->info[++p->top] = item;
}

char pop(STK *p) {
    if( p->top == -1 ) {
        printf("\n Stack Underflow");
        exit(1);
    }
    printf("\nItem popped %c", p->info [p->top]);
    return (p->info [p->top--]);
}
```

Example: **infix:** a+b*c\$d-(e-f/g)*h
 postfix: a b c d \$ * + e f g / - h * -

3: Write an algorithm to convert an infix expression into prefix or polish expression.

- STEP 1:** Insert a left parenthesis '(' at the beginning of the infix expression and push a right parenthesis ')' onto STACK.
- STEP 2:** Scan the infix expression from right to left and for each character repeat STEP 3 to STEP 6 until the STACK is empty.
- STEP 3:** If an operand is encountered, write it into the output string.
- STEP 4:** If a right parenthesis ')' is encountered, push it onto the STACK.
- STEP 5:** If an operator is encountered, repeatedly pop operators (if any) from STACK whose precedence is greater than that of encountered operator. Write those popped operators (if any) into output string. Then push the encountered operator onto STACK.
- STEP 6:** If a left parenthesis '(' is encountered, repeatedly pop operators (if any) from stack until a right parenthesis ')' is popped from the STACK. Write those popped operators (if any) into output string.
- STEP 7:** Reserved output string gives the required prefix expression.
- STEP 8:** Exit.

/* WRITE A PROGRAM TO CONVERT INFIX EXPRESSION INTO PREFIX OR POLISH EXPRESSION. */

```
# include <stdio.h>
# include <string.h>
# include <stdlib.h>
# define MAX 100

struct stack {
    char info[MAX];
    int top;
};
```



```
typedef struct stack STK;

void infix_to_prefix(char[], char[]);
int isoperator(char);
int precedence(char);
void push(STK *, char);
char pop(STK *);
void str_rev(char *);

int main(void) {
    char infix[MAX], prefix[MAX];
    printf("\nGive an infix expression: ");
    scanf("%s", infix);
    infix_to_prefix(infix, prefix);
    printf("\nPrefix expr. of %s is %s", infix, prefix);
    return 0;
}

void infix_to_prefix(char a[], char b[]) {
    STK stk;
    int i, j, l;
    char x;

    stk.top = -1;
    l = strlen(a);
    str_rev(a);
    a[l] = '(';
    push(&stk, '(');
    i = j = 0;

    while( stk.top != -1 ) {
        printf("\nScanning character %c", a[i]);
        if(a[i] == ')') {
            push(&stk, a[i++]);
        } else {
            if(isoperator(a[i])) {
                while(precedence( stk.info[stk.top]) > precedence(a[i])) {
                    b[j++] = pop(&stk);
                }
                push(&stk, a[i]);
            } else {
                if(a[i] == '(') {
                    while((x=pop(&stk)) != ')') {
                        b[j++] = x;
                    }
                } else {
                    b[j++] = a[i];
                }
            }
        }
        i++;
    }
}
```

```
    }
    b[j] = '\0';
    str_rev(b);
    a[l] = '\0';
    str_rev(a);
}

int isoperator( char x ) {
    return (x=='+' || x=='-' || x=='*' || x=='/' || x=='$');
}

int precedence(char op) {
    switch(op)
    {
        case '+':
        case '-':
            return (1);
        case '*':
        case '/':
            return (2);
        case '$':
            return (3);
        case ')':
            return (0);
        default:
            printf("\nInvalid operator %c", op );
            exit(1);
    }
}

void push(STK *p, char item) {
    printf("\nItem pushed %c", item);
    if( p->top == MAX-1 ) {
        printf("\nStack Overflow");
        return;
    }
    p->info[++p->top] = item;
}

char pop(STK *p) {
    if( p->top == -1 ) {
        printf("\nStack Underflow");
        exit(1);
    }
    printf("\nItem popped %c", p->info [p->top]);
    return (p->info[p->top--]);
}

void str_rev(char *str) {
    int begin, end, count = 0;
    count = strlen(str);
```

```
end = count - 1;
for (begin = 0; begin < end; begin++, end--) {
    str[begin] = str[begin] + str[end];
    str[end] = str[begin] - str[end];
    str[begin] = str[begin] - str[end];
}
}
```

Example: **infix:** a+b*c\$d-(e-f/g)*h
 prefix: - + a * b \$ c d * - e / f g h

Prefix to Infix Conversion

Algorithm for Prefix to Infix:

- A. Read the Prefix expression in reverse order (from right to left)
- B. If the symbol is an operand, then push it onto the Stack
- C. If the symbol is an operator, then pop two operands from the Stack
 - a. Create a string by concatenating the two operands and the operator between them. string = (operand1 + operator + operand2)
 - b. And push the resultant string back to Stack
- D. Repeat the above steps until the end of Prefix expression.
- E. At the end stack will have only 1 string i.e. resultant string