Show Course Contents

Overview Q&A Downloads Announcements

Coding Infix to Postfix in C using Stack

We saw earlier how infix expressions can be converted to their other equivalents manually. But when it came to automating the process, we took a different path. We used stacks to take hold of the operators we encountered in the expression. We followed an algorithm to convert an infix expression to its postfix equivalent, which in short, said:

1. We create a string variable that will hold our postfix expression. We start moving from the left to the right. And the moment we receive an operand, we concatenate it to the postfix string. And whenever we encounter an operator, we proceed with the following steps:

- Keep in account the operator and its relative precedence.
- If either the stack is empty or its topmost operator has lower relative precedence, push this operator-precedence pair inside the stack.
- Else, keep popping operators from the stack and concatenate it to the postfix expression until the topmost operator becomes weaker in precedence relative to

the current operator.

2. If you reach the EOE, pop every element from the stack, if there is any, and concatenate them as well. And there, you'll have your postfix expression.

Let us now see the program pursuing the conversion. I have attached the snippets alongwith. Keep checking them while you understand the codes.

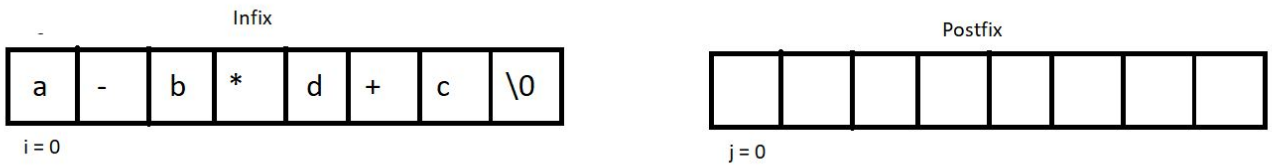
Understanding the program for infix to postfix conversion:

1. First of all, create a character pointer function *infixToPostfix* since the function has to return a character array. And now pass into this function the given expression, which is also a character pointer.

2. Define a struct stack pointer variable *sp.* And give it the required memory in the heap. Create the instance. It's safe to assume that a struct stack element and all its basic operations, *push, pop, etc.,* have already been defined. You better copy everything from the stack tutorial.

3. Create a character array/pointer *postfix,* and assign it sufficient memory to hold all the characters of the infix expression in the heap.

4. Create two counters, one to traverse through the *infix* and another to traverse and insert in the *postfix*. Refer to the illustration below, which describes the initial conditions.



5. Run a while loop until we reach the EOE of the *infix*. And inside that loop, check if the current index holds an operator, and if it's not, add that character into the postfix and increment both the counters by 1. And if it does hold an operator, call another function that would check if the precedence of the *stackTop* is less than the precedence of the current operator. If yes, push it inside the stack. Else, pop the *stackTop,* and add it back into the postfix. Increment j by 1.

```
char* infixToPostfix(char* infix){
    struct stack * sp = (struct stack *) malloc(sizeof(struct stack));
    sp->size = 10;
    sp->top = -1;
```

```
    sp->arr = (char *) malloc(sp->size * sizeof(char));
    char * postfix = (char *) malloc((strlen(infix)+1) * sizeof(char))
    int i=0; // Track infix traversal
    int j = 0; // Track postfix addition
    while (infix[i]!='\0')
    {
        if(!isOperator(infix[i])){
            postfix[j] = infix[i];
            j++;
            i++;
        }
        else{
            if(precedence(infix[i])> precedence(stackTop(sp))){
                push(sp, infix[i]);
                i++;
            }
            else{
                postfix[j] = pop(sp);
                j++;
            }
        }
    }
    while (!isEmpty(sp))
    {
        postfix[j] = pop(sp);
        j++;
    }
    postfix[j] = '\0';
    return postfix;
}
```

Code Snippet 1: Creating the function infixToPostfix

6. It's now time to create the two functions to make this conversion possible. *isOperator & precedence* which checks if a character is an operator and compares the precedence of two operators respectively.

7. Create an integer function *isOperator* which takes a character as its parameter and returns 2, if it's an operator, and 0 otherwise.

```
int isOperator(char ch){
    if(ch=='+' || ch=='-' ||ch=='*' || ch=='/')
        return 1;
    else
        return 0;
}
```

Code Snippet 2: Creating the function isOperator

8. Create another integer function *precedence,* which takes a character as its parameter, and returns its relative precedence. It returns 3 if it's a '/' or a '*'. And 2 if it's a '+' or a '-'.

9. If we are still left with any element in the stack at the end, pop them all and add them to the *postfix.*

```
int precedence(char ch){
    if(ch == '*' || ch=='/')
        return 3;
    else if(ch == '+' || ch=='-')
        return 2;
    else
        return 0;
}
```

Code Snippet 3: Creating the function *precedence*

And we have successfully finished writing the codes.

Here is the whole source code:

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

struct stack
```

```
{
    int size;
    int top;
    char *arr;
};

int stackTop(struct stack* sp){
    return sp->arr[sp->top];
}

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, char val){
    if(isFull(ptr)){
```

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

char pop(struct stack* ptr){
    if(isEmpty(ptr)){
        printf("Stack Underflow! Cannot pop from the stack\n");
        return -1;
    }
    else{
        char val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}
int precedence(char ch){
    if(ch == '*' || ch=='/')
        return 3;
    else if(ch == '+' || ch=='-')
        return 2;
    else
        return 0;
}

int isOperator(char ch){
    if(ch=='+' || ch=='-' ||ch=='*' || ch=='/')
        return 1;
    else
        return 0;
}
char* infixToPostfix(char* infix){
    struct stack * sp = (struct stack *) malloc(sizeof(struct stack));
```

```
    sp->size = 10;
    sp->top = -1;
    sp->arr = (char *) malloc(sp->size * sizeof(char));
    char * postfix = (char *) malloc((strlen(infix)+1) * sizeof(char))
    int i=0; // Track infix traversal
    int j = 0; // Track postfix addition
    while (infix[i]!='\0')
    {
        if(!isOperator(infix[i])){
            postfix[j] = infix[i];
            j++;
            i++;
        }
        else{
            if(precedence(infix[i])> precedence(stackTop(sp))){
                push(sp, infix[i]);
                i++;
            }
            else{
                postfix[j] = pop(sp);
                j++;
            }
        }
    }
    while (!isEmpty(sp))
    {
        postfix[j] = pop(sp);
        j++;
    }
    postfix[j] = '\0';
    return postfix;
}
int main()
{
```

Code Snippet 4: Source code for the function infixToPostfix

We now need to check the function for some expressions to see if it works.

```
char * infix = "x-y/z-k*d";
printf("postfix is %s", infixToPostfix(infix));
```

Code Snippet 5: Calling the function infixToPostfix

```
postfix is xyz/-kd*-
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above program

So, yes, we could automate the process of conversion through some codes. This must surely be easy for you after having a strong grip on the basics. If you missed any of these concepts, do mind checking them out.

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial, where we'll start exploring another data structure called **Queue**. Till then, keep coding.

Previous Next

CodeWithHarry