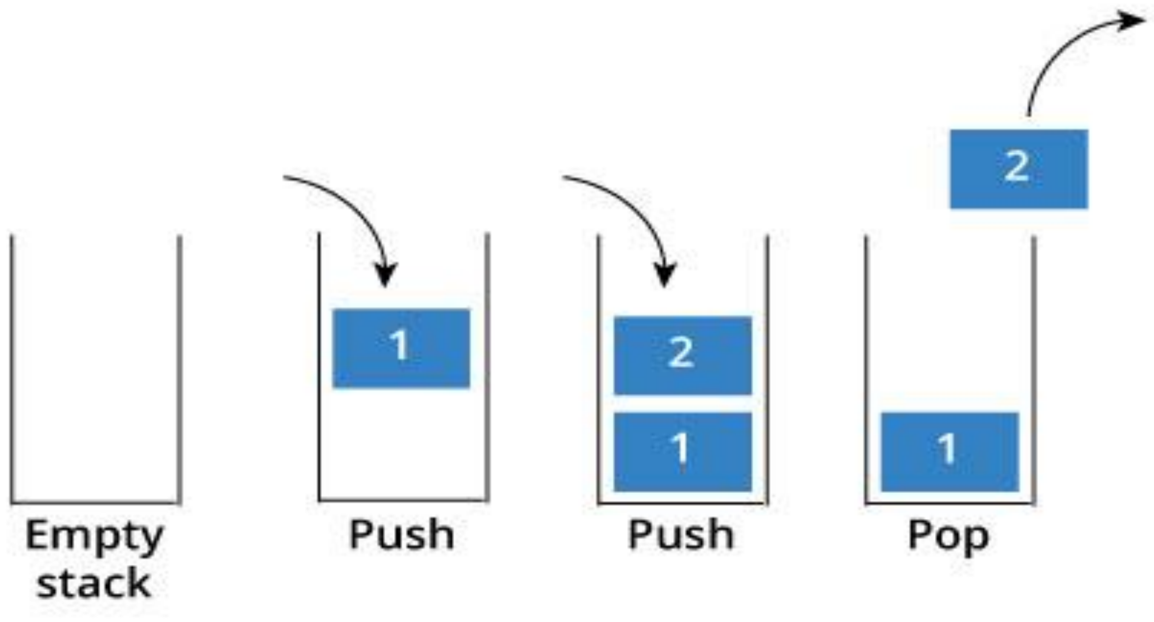
Unit 2: Stack (4)



Presented By : Tekendra Nath Yogi

Tekendranath@gmail.com

College Of Applied Business And Technology

Contents

- Basic Concept of Stack
- Stack as an ADT
- Stack Operations:
- Stack Applications (Bracket matching).
- Conversion from infix to postfix/prefix expression
- Evaluation of postfix/prefix expressions

Introduction

- A Stack (pushdown list) is a linear data structure, which is based on LIFO principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP.

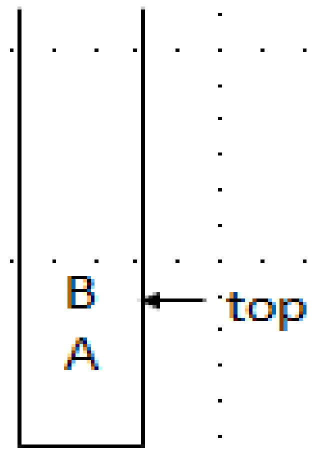


Figure: Stack containing Stack items

- Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

Contd…

- **For Example:** If we add the elements A, B, C, D, E to the stack, in that order, then E is the first element we delete from the stack

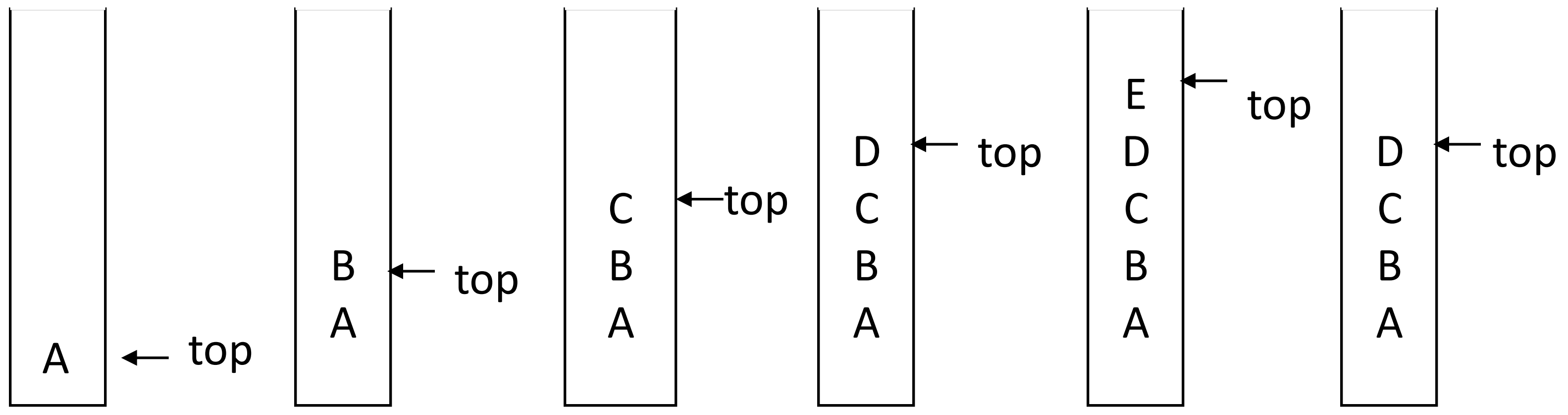


Fig: motion picture of Push and pop operations on a stack

Contd…

- **Basic features of Stack**
  - Stack is an **ordered list** of **similar data type**.
  - Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).
  - It has only one pointer **TOP** that points the last or top most element of Stack.
  - Insertion and Deletion in stack can only be done from top only.
  - Insertion in stack is also known as a **PUSH operation**.
  - Deletion from stack is also known as **POP operation** in stack.
  - Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

Contd…

- **Applications of Stack**
  - **When function is called**

    When a function is called, the function is called last will be completed first. It is the property of stack. There is a memory area, specially reserved for this stack.
  - **To reverse a string**

    A string can be reversed by using stack. The characters of string pushed on to the stack till the end of the string. The characters are popped and displays. Since the end character of string is pushed at the last, it will be printed first.

Contd…

- **To solve Bracket Matching Problem (delimiter/parenthesis matching problem):** Ensures that pairs of brackets are properly matched. For every char read if open bracket then push onto stack, if close bracket, then return & remove most recent item from the stack, if doesn't match then return error, if non-bracket, skip the char read
- **Conversion of polish notations**

  There are three types of notations:

  – **Infix notation** - Operator is between the operands : x + y

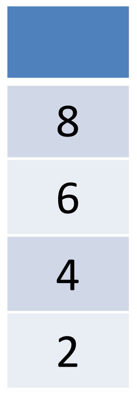
  – **Prefix notation** - Operator is before the operands : + xy

  – **Postfix notation** - Operator is after the operands : xy +

Stack as an ADT

- **Data:** An ordered sequence of elements of particular type T.

  E.g., S=



Stack S is an ordered sequence of integer type elements.

- **Operations:**
  - createStack(s)
  - Push(s)
  - Pop(s)
  - Peek(s)
  - isFull(s)
  - isEmpty(s)

Stack as an ADT

- **Alternatively we can define stack as an ADT as shown below:**
- **Data:** a finite ordered list with zero or more elements of type T.
- **Operations:**

```
 for all stack ∈ Stack, item ∈ element, max_stack_size ∈ positive integer
Stack createStack(max_stack_size) =
        create an empty stack whose maximum size is max_stack_size
Boolean isFull(stack, max_stack_size) =
        if (number of elements in stack == max_stack_size) return TRUE
        else     return FALSE
Stack push(stack, item) =
        if (IsFull(stack)) stack_full
        else insert item into top of stack and return
Boolean isEmpty(stack) =
            if(stack == CreateStack(max_stack_size))
                          return TRUE
            else return FALSE
Element pop(stack) =
            if(IsEmpty(stack)) return
            else remove and return the item on the top of the stack.
Element Peek(stack)=
      if(isEmpty(stack)== TRUE) then return
      else return the top element of the stack.
```

Stack operations

- On stack data structure following operations can be used:
  - **createStack(S):** used to create empty stack S.
  - **Push(S, x):** used to add new element x into stack x.
  - **Pop(S):** used to remove element at the top of stack.
  - **Peek(S):** used to retrieve top element of a stack S without changing stack.
  - **isFull(S):** Used to determine whether stack S is full or not. Return true if S is full; return false otherwise.
  - **isEmpty(S):** Used to determine whether stack S is empty or not. Return true if S is empty; return false otherwise.

Contd…

- **createStack Operation:**
  - This operation is used to create new stack of desired size called Max size stack.
  - One possible way to create a stack is create array of size Max with a top variable to indicate top of stack in the array.
  - Initially when the stack is created its top=-1.
  - If we add a new element into the stack, stack top is incremented by 1.
  - If stack contains one element then top =0.
  - If stack contains Max number of elements then top=Max-1.

Contd…

- **Push operations:**
  - The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack.
  - To insert an element, we must first check if TOP=MAX–1. If the condition is false, then we increment the value of TOP and store the new element at the top position of stack.
  - If **TOP=MAX–1** is true, then the stack is full and no more insertions can be done. If an attempt is made to insert a value in a stack that is already full, an **OVERFLOW** message is printed.

Contd…

- **Algorithm for Push Operation:**

```
push( )
  {
        If (top= = Max-1)
        {
              Print “Overflow”
              Exit.
        }
        Else
        {
           set top =top +1
           set stack [top]= value
        }
  }
```

Contd…

- **Pop operation:**
  - The pop operation is used to delete the topmost element from the stack.
  - To delete the topmost element, we first check if TOP=NULL=-1. If the condition is false, then we decrement the value pointed by TOP.
  - If TOP=NULL=-1 is true, then it means the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.

Contd…

- **Algorithm for pop Operation:**

```
pop( )
 {
        If (top= = -1)
        {
              Print “Underflow”
              Exit. Or return -1
        }
        Else
        {
              Value=stack[top]
              set top =top -1
              Print “value” or return value
        }
 }
```

Contd…

- **Peek operation:**
  - Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
  - However, the Peek operation first checks if the stack is empty, i.e., if TOP = =-1, then an appropriate message is printed, else the value is returned.
  - **Algorithm for isFull Operation:**

```
peek( )
{
      if( top= = -1)
            print “Underflow!”;
      else
            return  element at the top of stack ;
}
```

Contd…

- **isFull Operation:** Used to check whether a stack is full or not, if the stack is full this operation returns true; returns false otherwise. If the stack is full then the top =Max-1. so, no new item can be added on the stack. This situation is known as overflow.
- **Algorithm for isFull Operation:**

```
isFull( )
{
    if( top==Max-1)
        print " stack is full" or return 1;
      else
      print "stack is not full" or return 0;
}
```

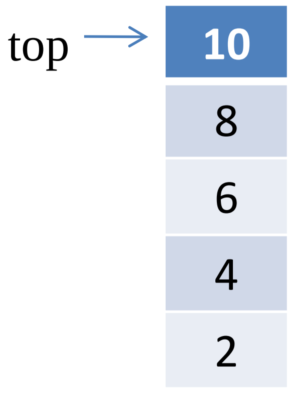


Figure: Full stack

Contd…

- **isEmpty Operation:** Used to check whether a stack is empty or not, if the stack is empty this operation returns true; returns false otherwise. If the stack is empty then the top ==-1. so, no item can be removed from the stack. This situation is known as underflow.

- **Algorithm for isFull Operation:**

```
isEmpty( )
{
    if( top= = -1)
        print " stack is empty" or return 1;
        else
            print " stack is not empty" or return 0;
}
```

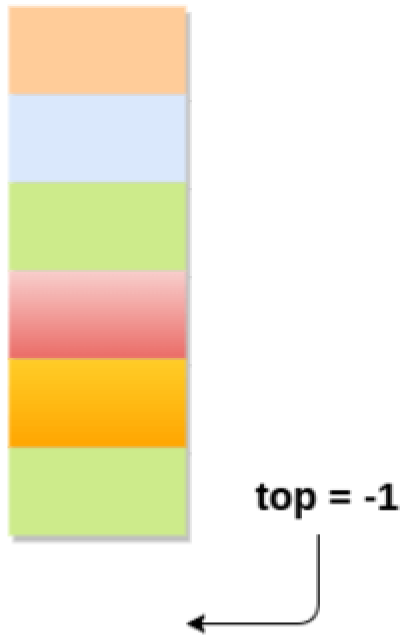
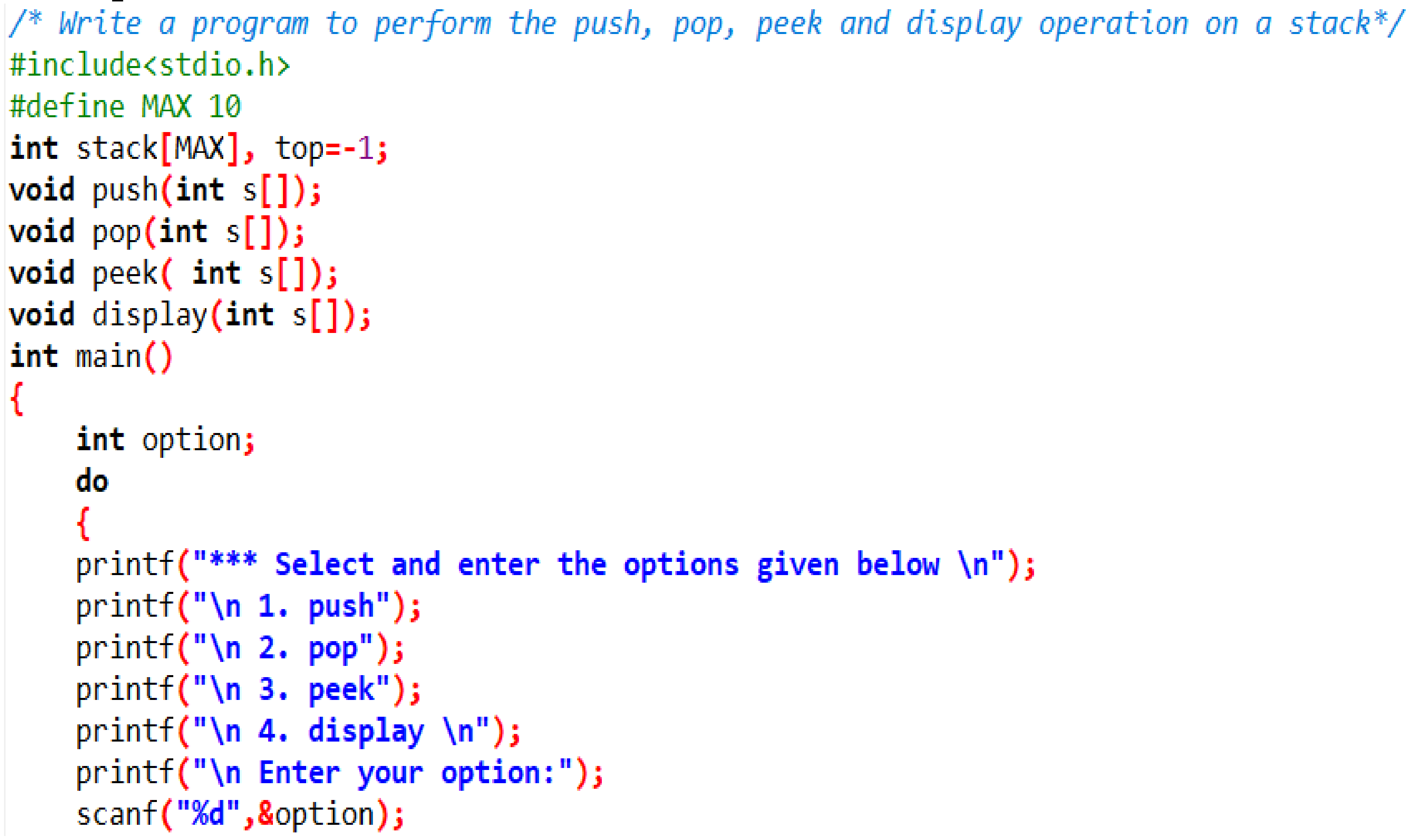


Figure: Empty stack

Array Implementation of stack

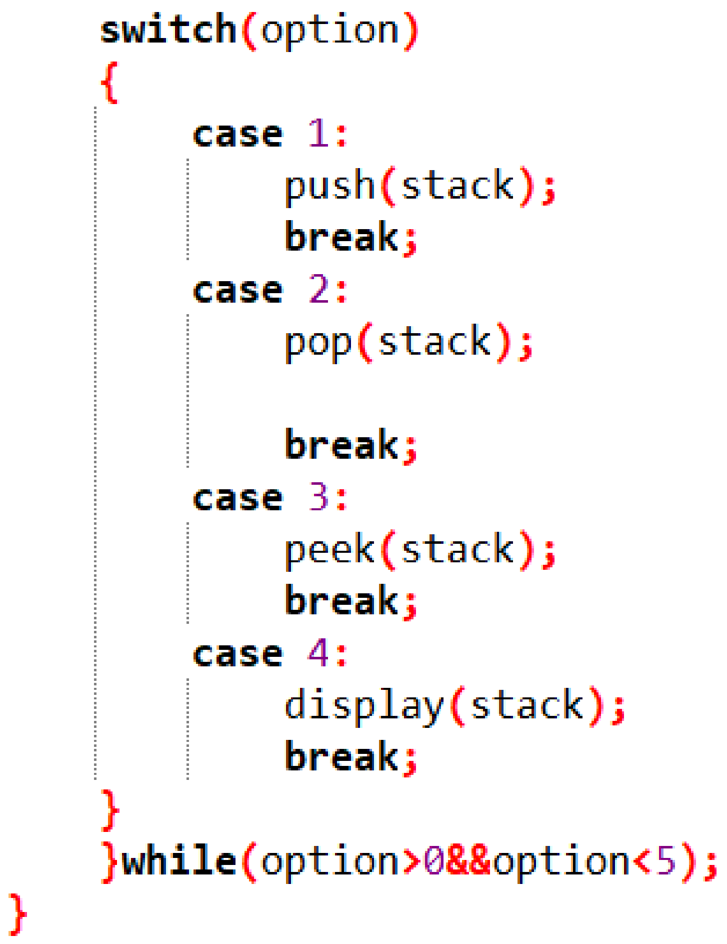
Example:



```
/* Write a program to perform the push, pop, peek and display operation on a stack*/
#include<stdio.h>
#define MAX 10
int stack[MAX], top=-1;
void push(int s[]);
void pop(int s[]);
void peek( int s[]);
void display(int s[]);
int main()
{
    int option;
    do
    {
    printf("*** Select and enter the options given below \n");
    printf("\n 1. push");
    printf("\n 2. pop");
    printf("\n 3. peek");
    printf("\n 4. display \n");
    printf("\n Enter your option:");
    scanf("%d",&option);
```

Contd…

- **Continue…..**

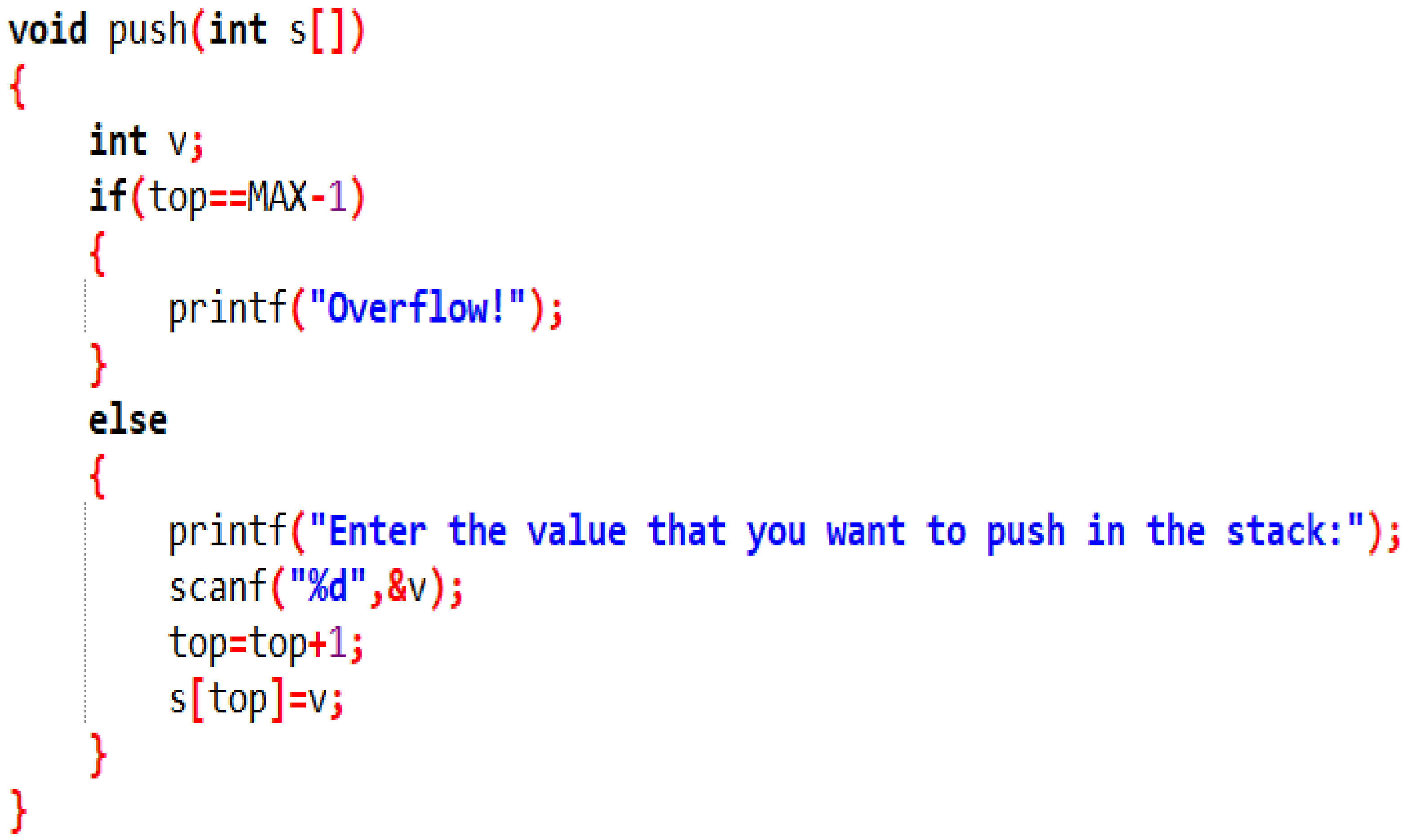


```
    switch(option)
    {
        case 1:
            push(stack);
            break;
        case 2:
            pop(stack);

            break;
        case 3:
            peek(stack);
            break;
        case 4:
            display(stack);
            break;
    }
    }while(option>0&&option<5);
}
```

Contd…

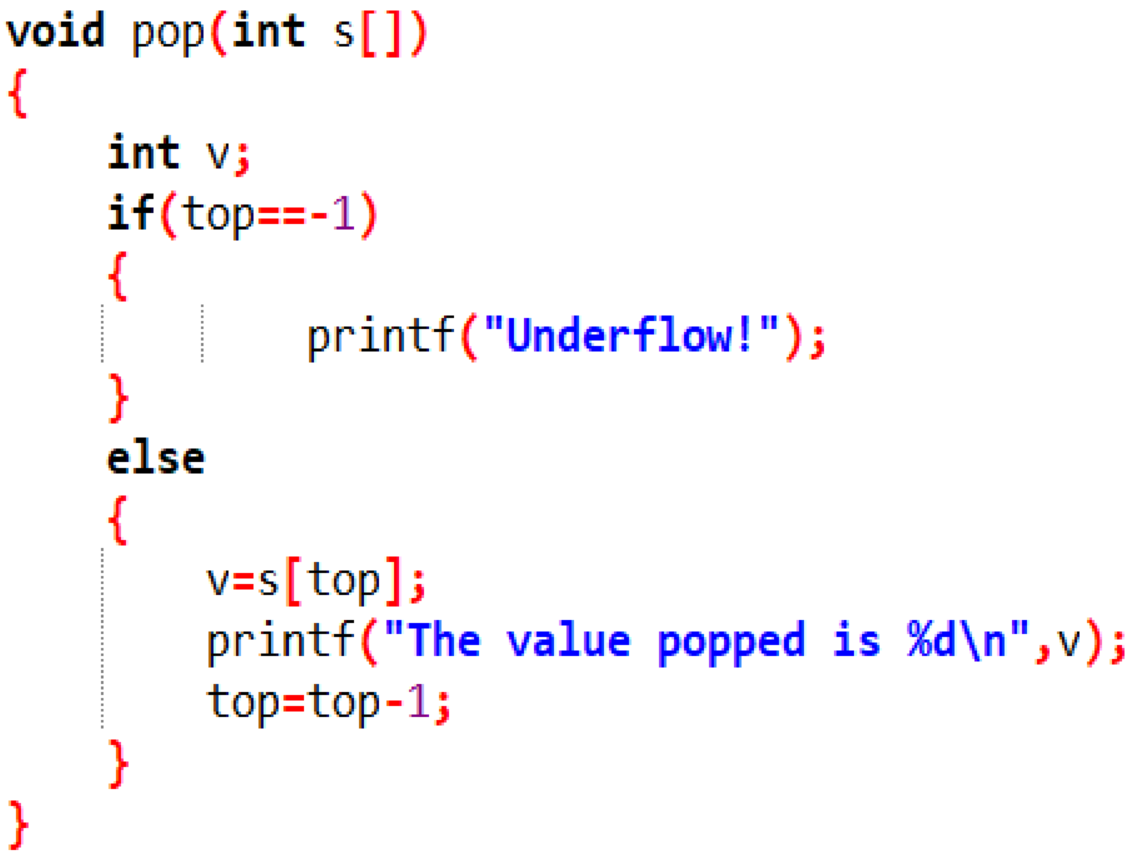
- **Continue…..**



```
void push(int s[])
{
    int v;
    if(top==MAX-1)
    {
        printf("Overflow!");
    }
    else
    {
        printf("Enter the value that you want to push in the stack:");
        scanf("%d",&v);
        top=top+1;
        s[top]=v;
    }
}
```

Contd…

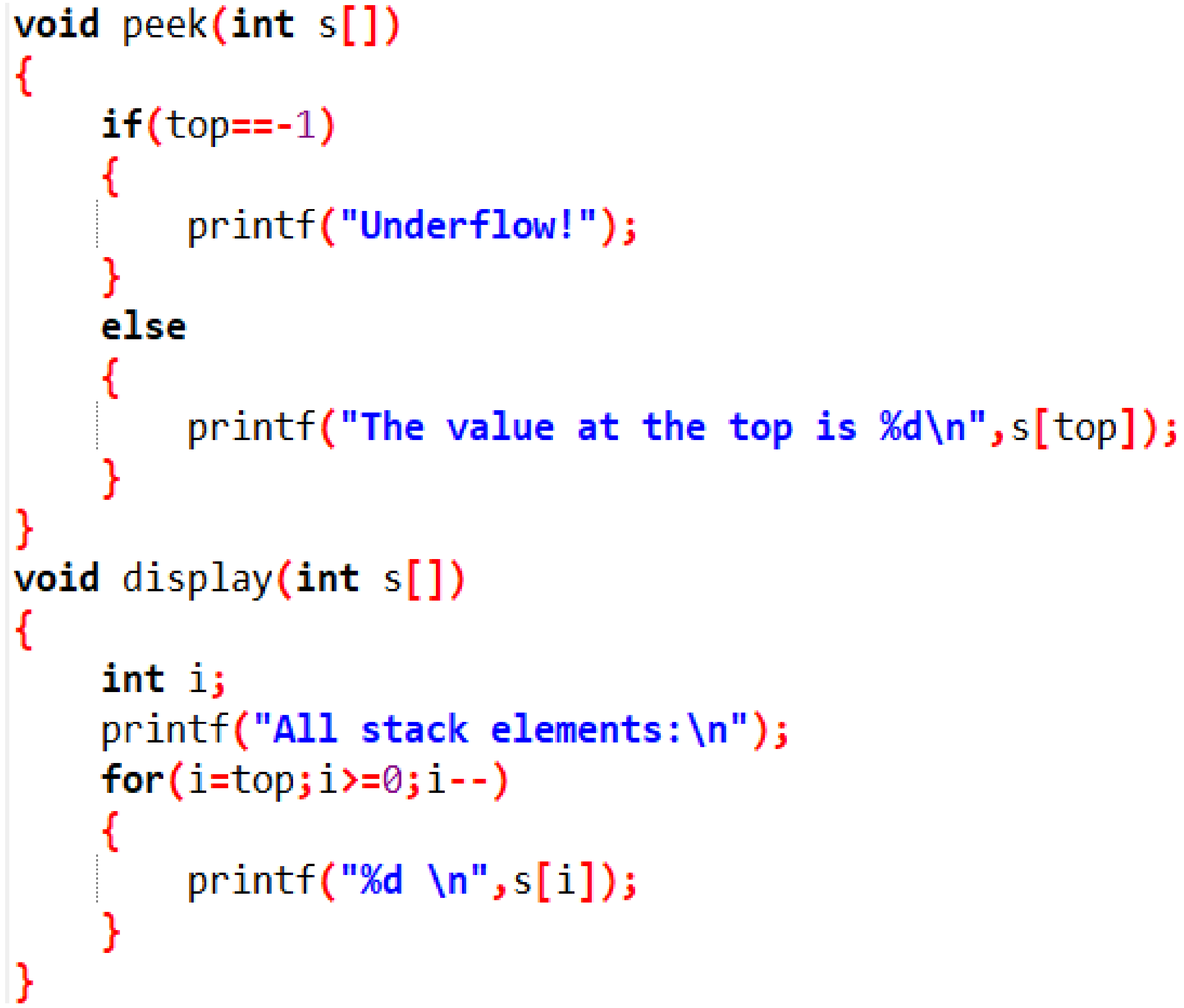
- **Continue…..**



```c
void pop(int s[])
{
    int v;
    if(top==-1)
    {
            printf("Underflow!");
    }
    else
    {
        v=s[top];
        printf("The value popped is %d\n",v);
        top=top-1;
    }
}
```

Contd…

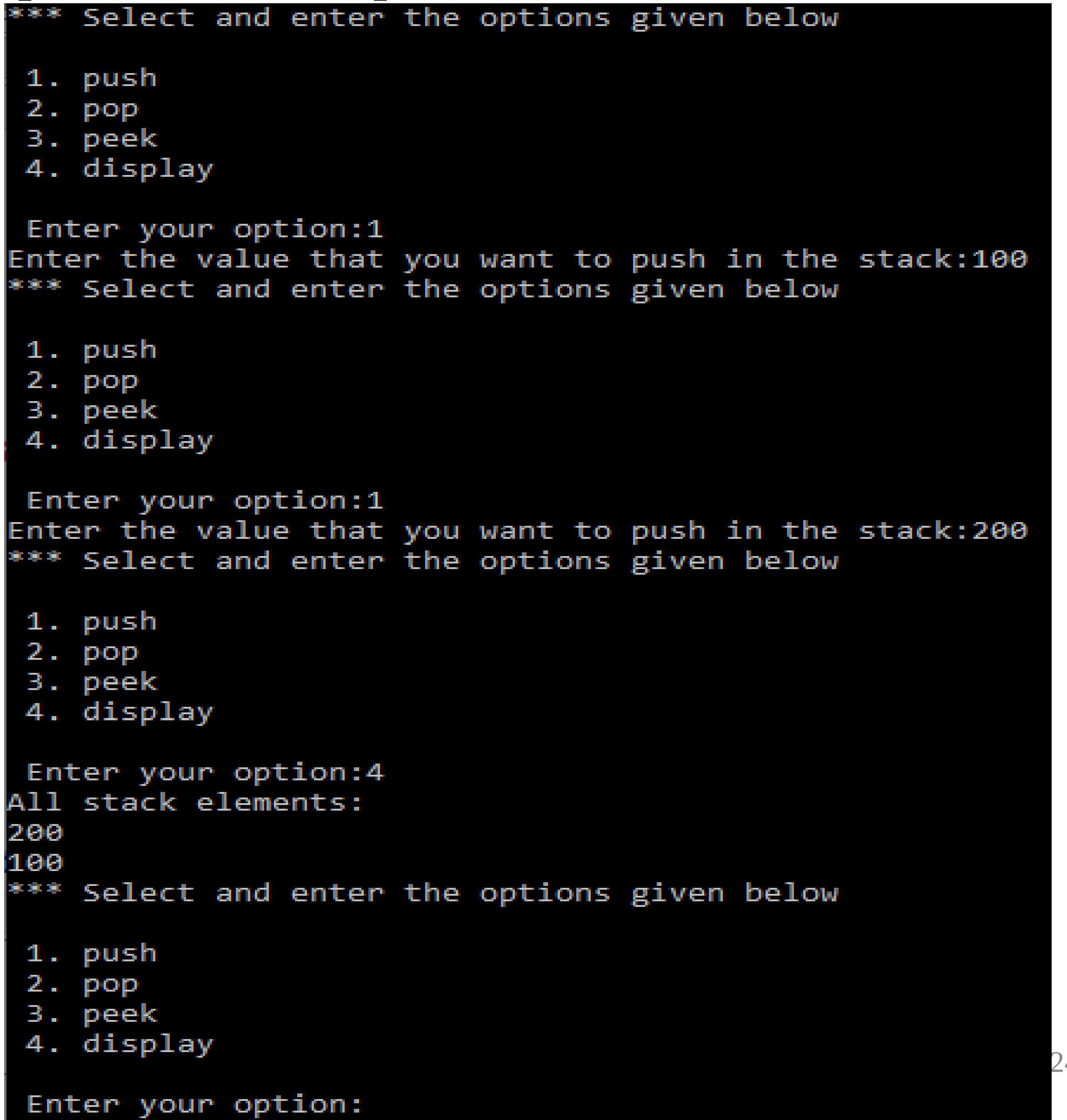
- **Continue…..**



```
void peek(int s[])
{
    if(top==-1)
    {
        printf("Underflow!");
    }
    else
    {
        printf("The value at the top is %d\n",s[top]);
    }
}
void display(int s[])
{
    int i;
    printf("All stack elements:\n");
    for(i=top;i>=0;i--)
    {
        printf("%d \n",s[i]);
    }
}
```

Contd…

- **Sample Run and Output:**



```
*** Select and enter the options given below

 1. push
 2. pop
 3. peek
 4. display

 Enter your option:1
Enter the value that you want to push in the stack:100
*** Select and enter the options given below

 1. push
 2. pop
 3. peek
 4. display

 Enter your option:1
Enter the value that you want to push in the stack:200
*** Select and enter the options given below

 1. push
 2. pop
 3. peek
 4. display

 Enter your option:4
All stack elements:
200
100
*** Select and enter the options given below

 1. push
 2. pop
 3. peek
 4. display

 Enter your option:
```

Stack Applications (Bracket matching)

- In a valid algebraic expression brackets must appear in a balanced fashion. Balanced brackets means that each opening symbols has corresponding closing symbol (pair) and the pairs of parentheses are properly nested.

  For Example: {a,(b+f[4])*3,d+f[5]}

  Examples of Invalid expressions :

  (..)..) // too many closing brackets

  (..(..) // too many open brackets

  [..(..]..) // mismatched brackets

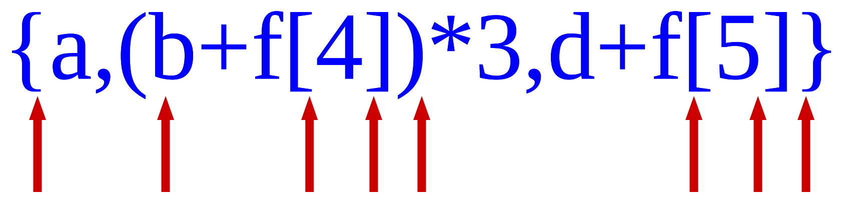
Contd…

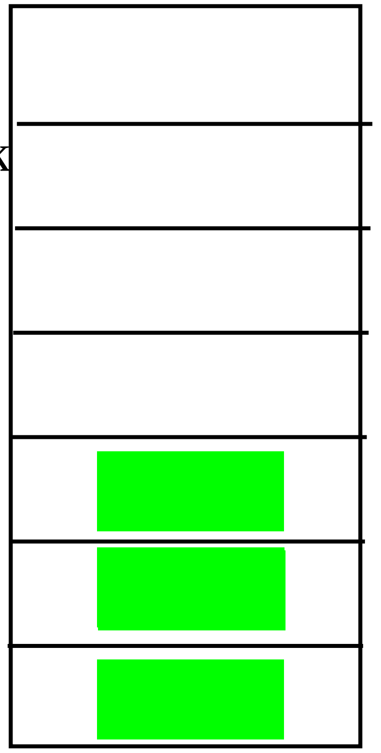
- **Bracket matching problem:** Decide whether the input string contains balanced brackets or not. Brackets are said to be balanced if they appear in a pair and also be in a proper order.
- Bracket matching problem can be **solved efficiently by using stack as**:
  - Read the input string left to right. During reading skip all non bracket symbols and consider all the bracket symbols as, the opening brackets in input string are pushed into the stack. If their corresponding matched closing bracket is found then pop the opening bracket from the stack and so on. At the end of input string if the stack is empty then the expression is valid otherwise the input string is invalid.

Contd…

- Initialize the stack to empty
- For every char read
  - if open bracket then push onto stack
  - if close bracket, then
    - return & remove most recent item from the stack
    - if doesn't match then flag invalid
  - if non-bracket, skip the char read

Example

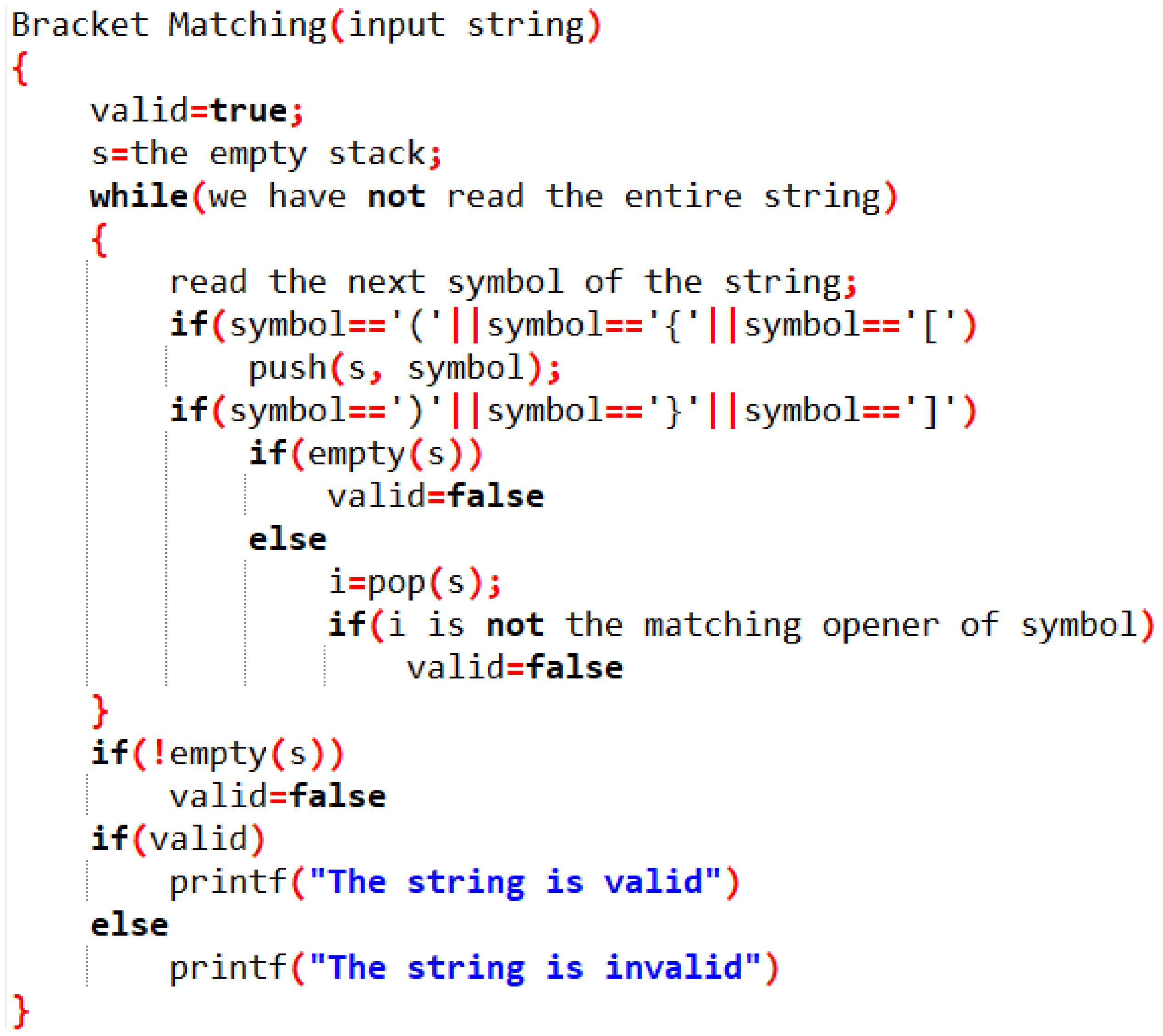




Stack

Contd…

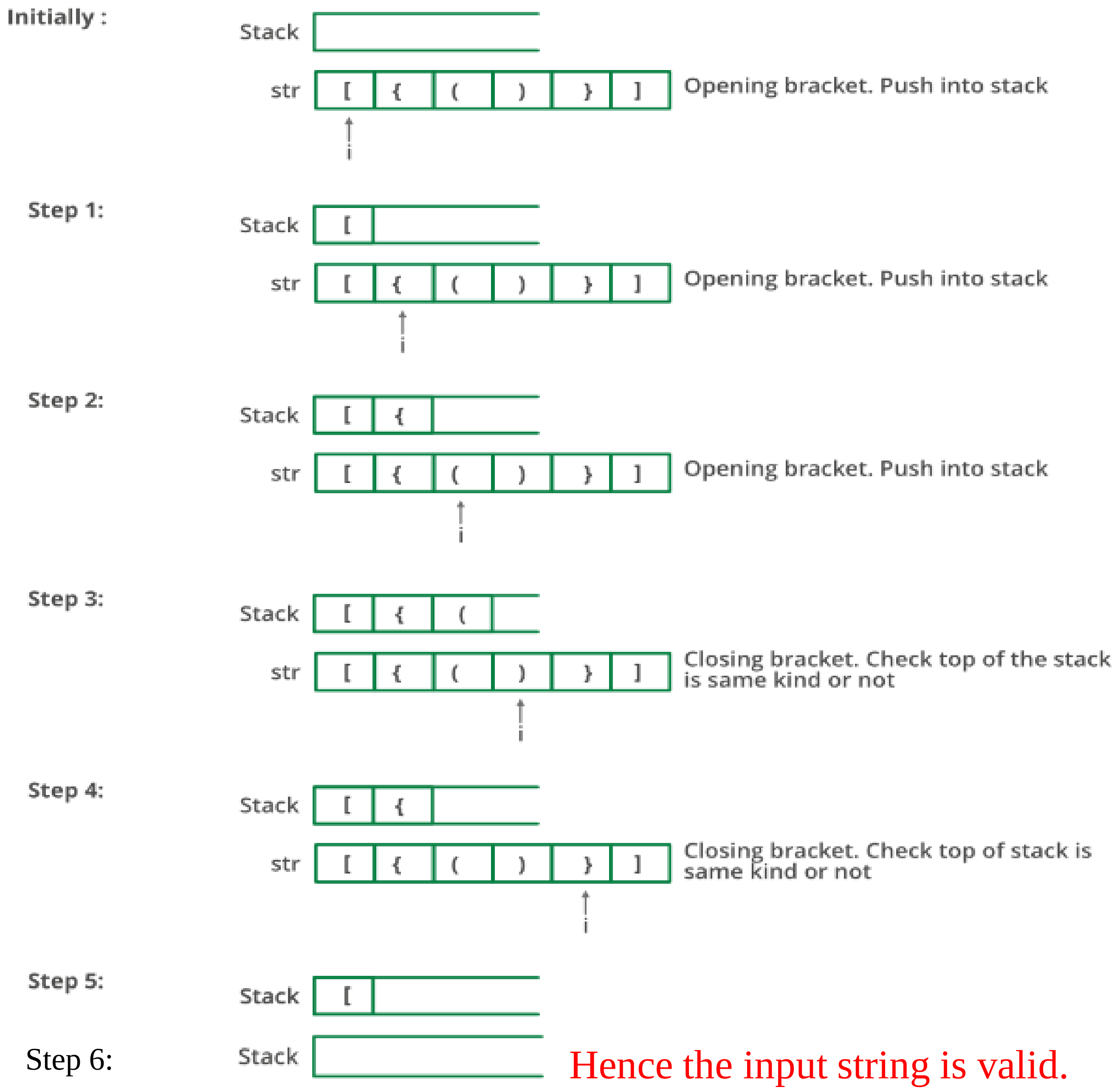
- **Algorithm for Bracket Matching:**



```
Bracket Matching(input string)
{
    valid=true;
    s=the empty stack;
    while(we have not read the entire string)
    {
        read the next symbol of the string;
        if(symbol=='('||symbol=='{'||symbol=='[')
            push(s, symbol);
        if(symbol==')'||symbol=='}'||symbol==']')
            if(empty(s))
                valid=false
            else
                i=pop(s);
                if(i is not the matching opener of symbol)
                    valid=false
    }
    if(!empty(s))
        valid=false
    if(valid)
        printf("The string is valid")
    else
        printf("The string is invalid")
}
```

Contd…

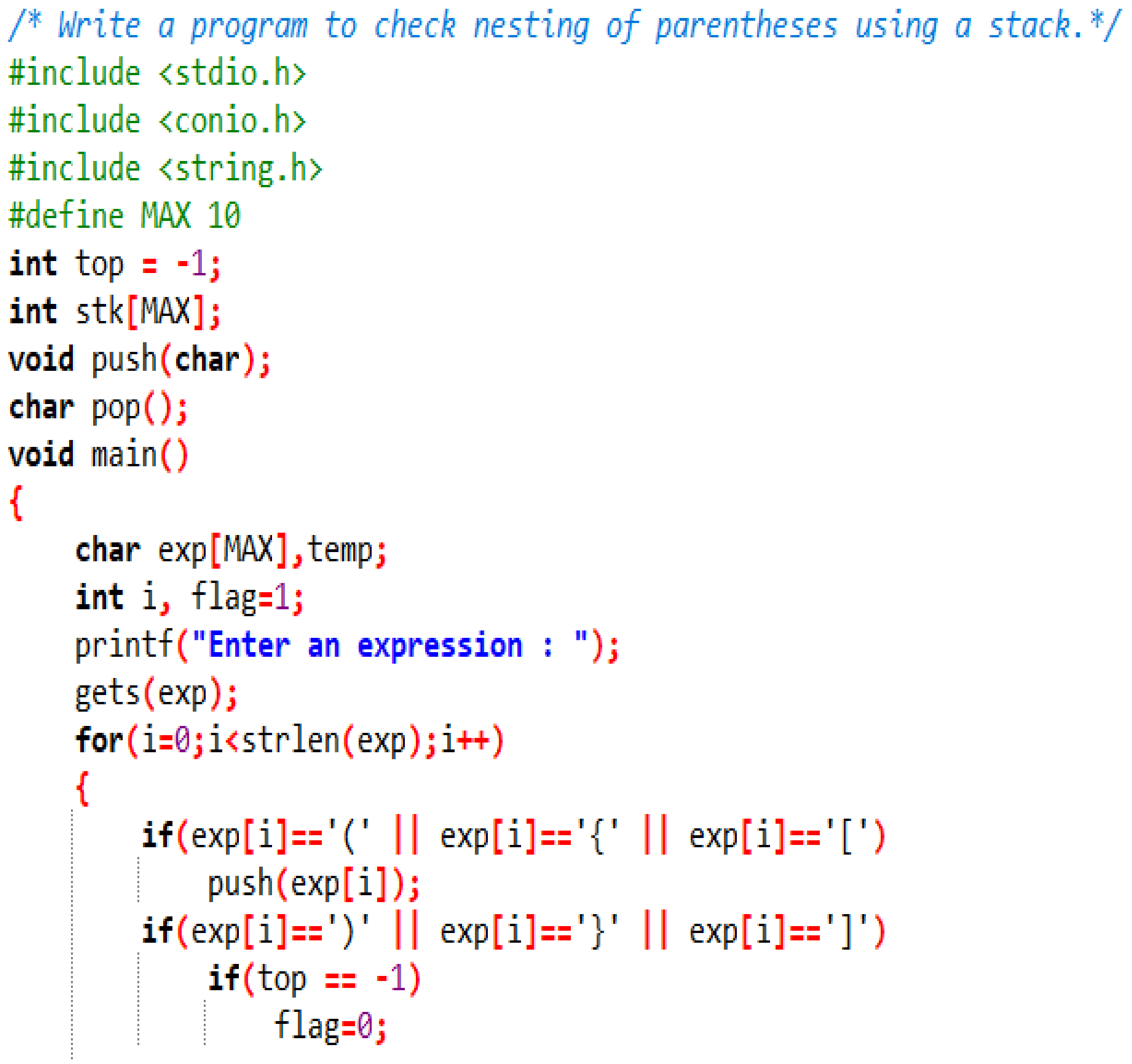
- **Example:**


Initially :
Stack
str [{ ( ) }]
Opening bracket. Push into stack
i
Step 1:
Stack [
str [{ ( ) }]
Opening bracket. Push into stack
i
Step 2:
Stack [{
str [{ ( ) }]
Opening bracket. Push into stack
i
Step 3:
Stack [{ (
str [{ ( ) }]
Closing bracket. Check top of the stack is same kind or not
i
Step 4:
Stack [{
str [{ ( ) }]
Closing bracket. Check top of stack is same kind or not
i
Step 5:
Stack [
Step 6:
Stack

Hence the input string is valid.

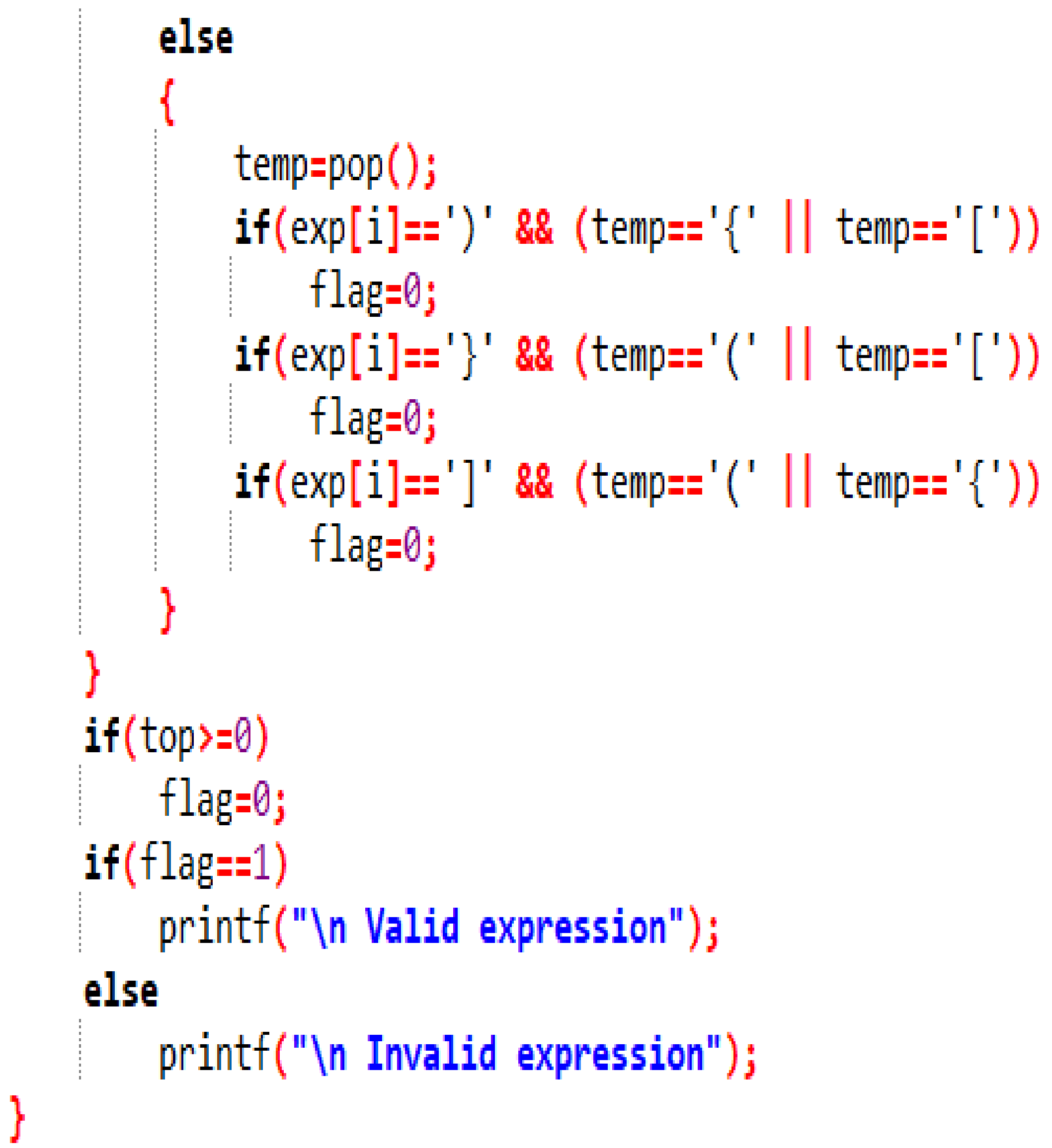
Contd…

- **Example: Write a program to check nesting of parentheses using a stack.**



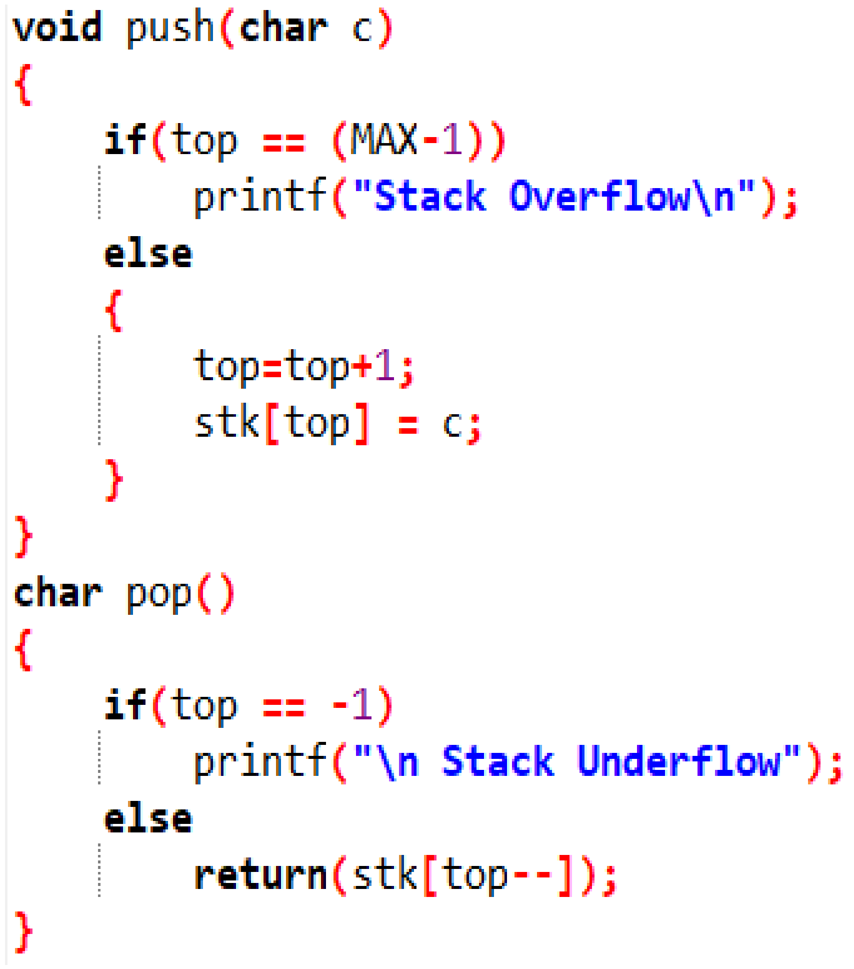
```
/* Write a program to check nesting of parentheses using a stack.*/
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 10
int top = -1;
int stk[MAX];
void push(char);
char pop();
void main()
{
    char exp[MAX],temp;
    int i, flag=1;
    printf("Enter an expression : ");
    gets(exp);
    for(i=0;i<strlen(exp);i++)
    {
        if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')
            push(exp[i]);
        if(exp[i]==')' || exp[i]=='}' || exp[i]==']')
            if(top == -1)
                flag=0;
```

Contd…



```
        else
        {
            temp=pop();
            if(exp[i]==')' && (temp=='{' || temp=='['))
                flag=0;
            if(exp[i]=='}' && (temp=='(' || temp=='['))
                flag=0;
            if(exp[i]==']' && (temp=='(' || temp=='{'))
                flag=0;
        }
    }
    if(top>=0)
        flag=0;
    if(flag==1)
        printf("\n Valid expression");
    else
        printf("\n Invalid expression");
}
```

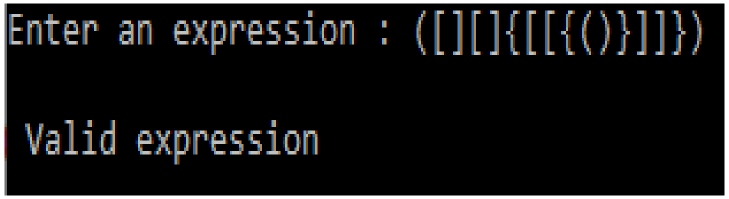
Contd…



```
void push(char c)
{
    if(top == (MAX-1))
        printf("Stack Overflow\n");
    else
    {
        top=top+1;
        stk[top] = c;
    }
}
char pop()
{
    if(top == -1)
        printf("\n Stack Underflow");
    else
        return(stk[top--]);
}
```

Contd…

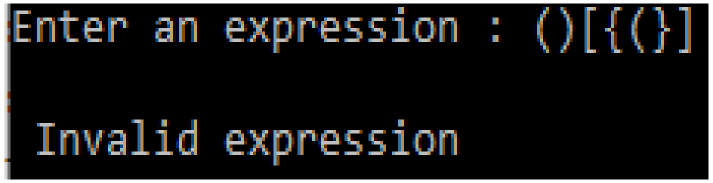
- **Sample Run and Output:**
  - **Run1 :**



```
Enter an expression : ([][]{[[{()}]]})

 Valid expression
```

  - **Run 2:**



```
Enter an expression : ()[{()}]

 Invalid expression
```

Evaluation And conversion of Arithmetic expressions

- Three different but equivalent notations for writing algebraic expressions are:
  - Infix: In infix notation operator is placed in between operands. e.g., A+B.
  - Prefix (Polish Notation or PN.): In prefix notation operator is placed before the operands. e.g., +AB.
  - Postfix (Reverse Polish Notation or RPN.): In postfix notation operator is placed after the operands. e.g., AB+.

Contd…

- **Infix notation:**
  - In infix notation operator is placed in between operands. e.g., A+B.
  - The order of evaluation of infix expression depends on the precedence and associativity rules of operators and also depends on the parenthesization present in the expression.
  - Example : let A =10 ,B= 20 and C=2 then the infix expression A+B*C result in 50 after evaluation, whereas the infix expression (A+B)*C result in 60 after evaluation.
  - **Advantage:** Easy for us.
  - **Disadvantage:** Inefficient for computer due to the precedence and associativity rules along with brackets.

Contd…

- **Postfix (Reverse Polish Notation or RPN):**
  - In postfix notation operator is placed after the operands. e.g., AB+.
  - The order of evaluation of postfix expression is always from left to right. Even brackets can not alter the order of evaluation.
  - A postfix expression does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated on the operands that are immediately left of them.
  - **Example :** let A =10, B= 20 and C=2 then the postfix expression ABC*+ result in 50 after evaluation always.
  - **Advantage:** Easy for computer.
  - **Disadvantage:** somewhat difficult for us.

Contd…

- **Prefix (Polish Notation or PN):**
  - In prefix notation operator is placed before the operands. e.g., +AB.
  - The order of evaluation of prefix expression is also always from left to right.
  - Like postfix, prefix expressions also do not follow the rules of operator precedence and associativity, and even brackets cannot alter the order of evaluation.
  - While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator.
  - **Example :** let A =10 , B= 20 and C=2 then the prefix expression +A*BC result in 50 after evaluation always.
  - **Advantage:** Easy for computer.
  - **Disadvantage:** some what difficult for us.

Conversion from infix to postfix expression

- **Algorithm to convert an infix notation to postfix notation:**

Step 1: Add “)” to the end of the infix expression

Step 2: Push “(“ on to the stack

Step 3: Repeat until each character in the infix expression is scanned

- **IF a “(“ is encountered, push it on the stack**
- **IF an operand is encountered, add it postfix expression.**
- **IF a “)” is encountered, then**

  a. Repeatedly pop operators from stack and add it to the postfix expression until a “(“ is encountered.

  b. Discard the “(“ . That is, remove the “(“ from stack and do not add it to the postfix expression
- **IF an operator o is encountered, then**

  a. Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has a higher precedence than **o.**

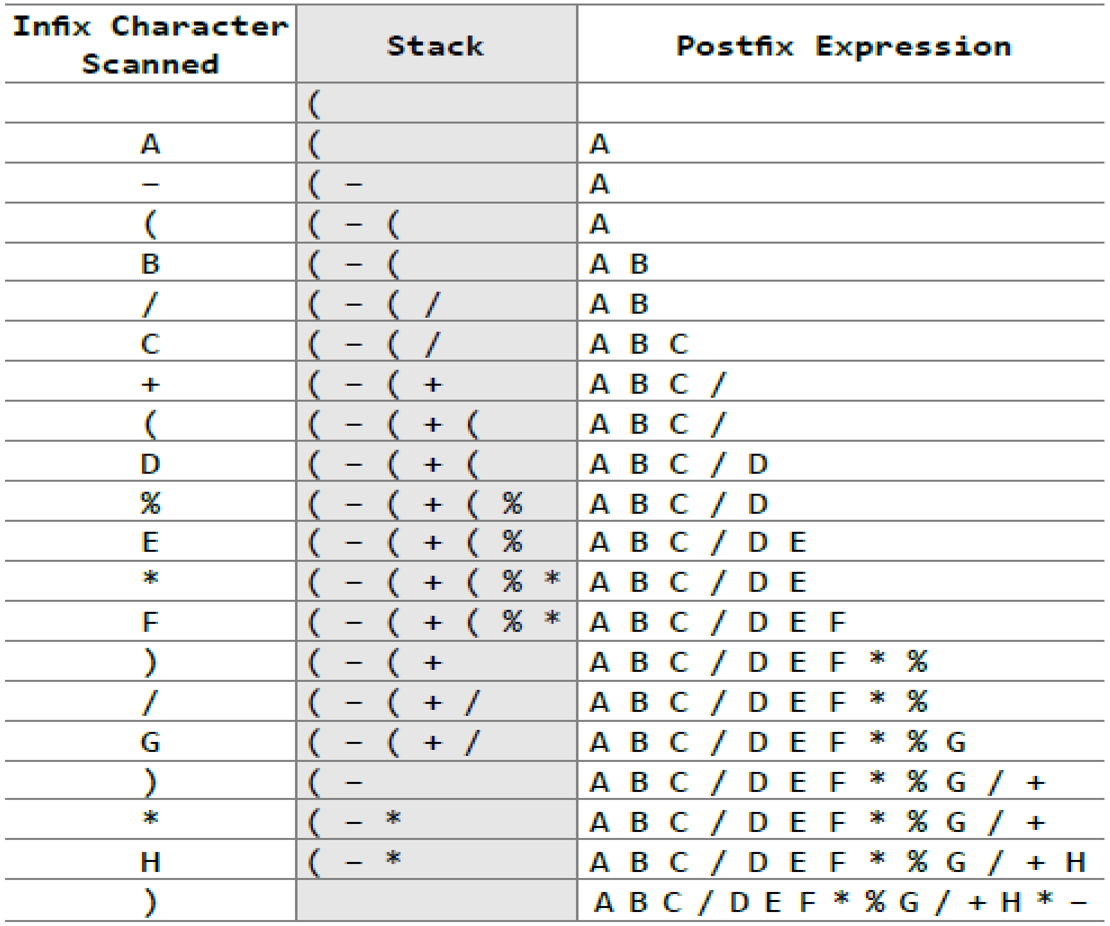
  b. Push the operator o to the stack

Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty

Step 5: EXIT

Contd…

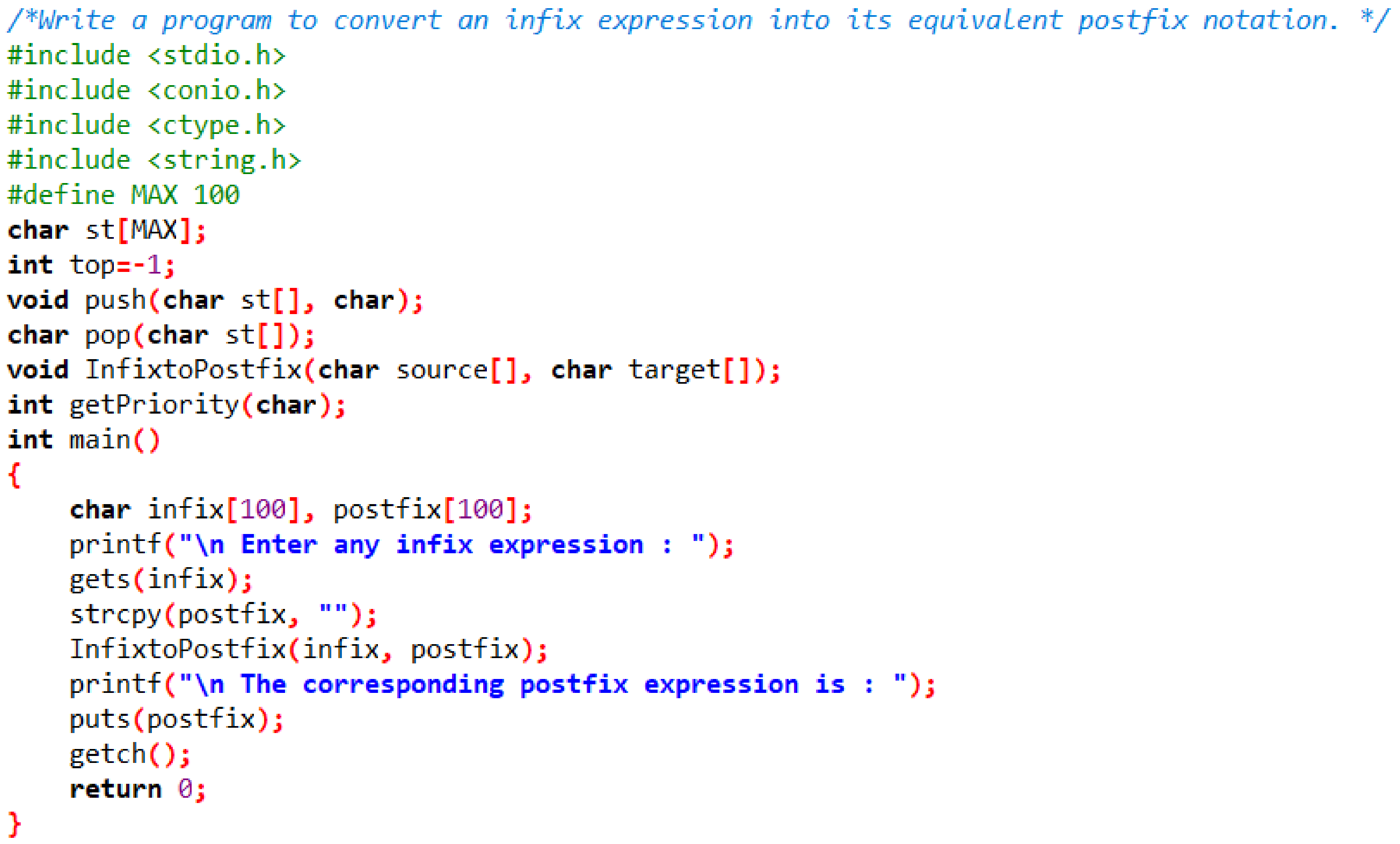
- **Example:** Convert the following infix expression **A – (B / C + (D % E * F) / G)* H** into postfix expression using the algorithm to convert infix to postfix notation.
- **Solution:**



Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
–	(–	A
(	(– (	A
B	(– (	A B
/	(– (/	A B
C	(– (/	A B C
+	(– (+	A B C /
(	(– (+ (	A B C /
D	(– (+ (	A B C / D
%	(– (+ (%	A B C / D
E	(– (+ (%	A B C / D E
*	(– (+ (% *	A B C / D E
F	(– (+ (% *	A B C / D E F
)	(– (+	A B C / D E F * %
/	(– (+ /	A B C / D E F * %
G	(– (+ /	A B C / D E F * % G
)	(–	A B C / D E F * % G / +
*	(– *	A B C / D E F * % G / +
H	(– *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * –

Contd…

- **Program: Write a program to convert an infix expression into its equivalent postfix notation.**

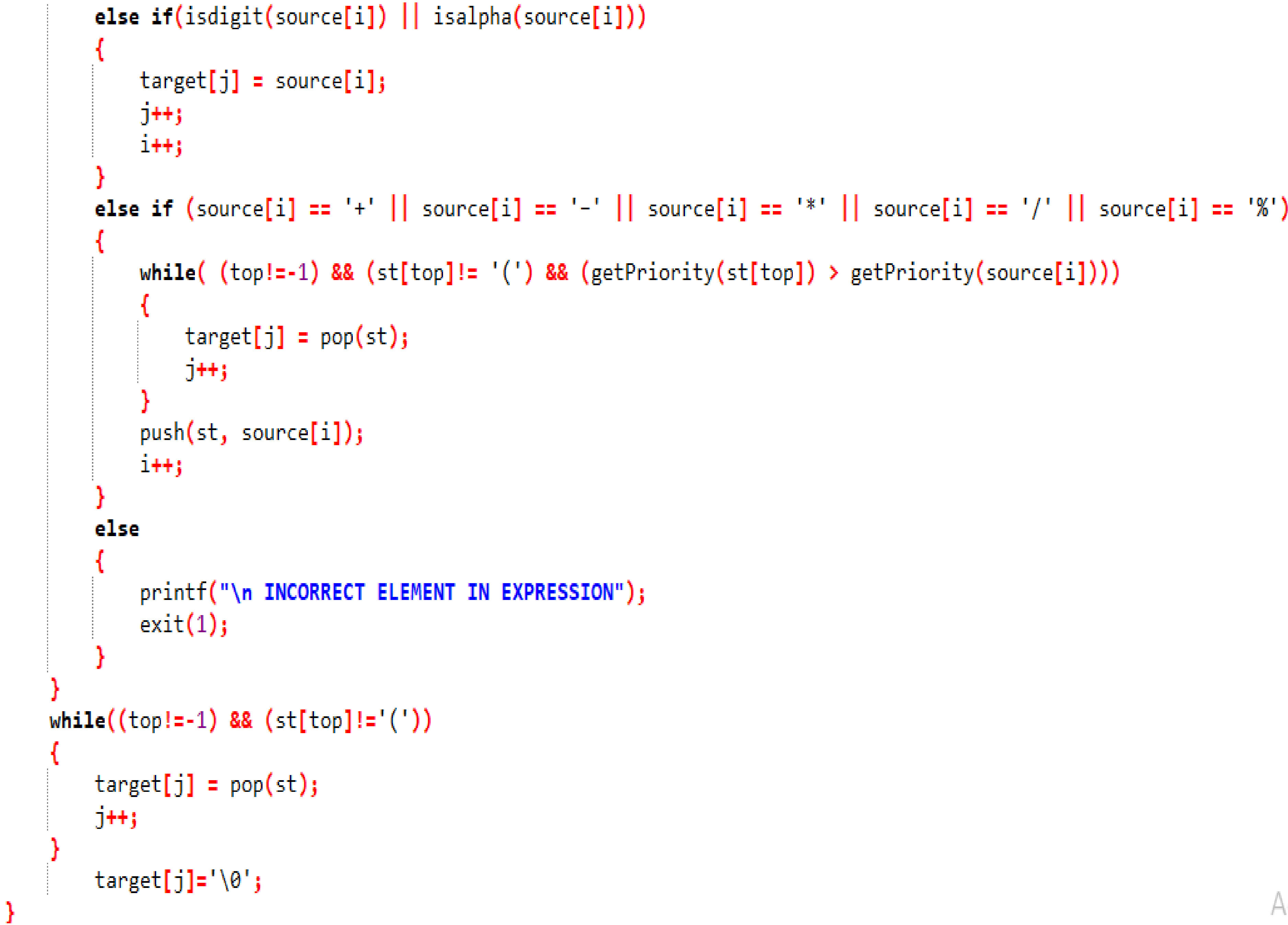


```
/*Write a program to convert an infix expression into its equivalent postfix notation. */
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>
#define MAX 100
char st[MAX];
int top=-1;
void push(char st[], char);
char pop(char st[]);
void InfixtoPostfix(char source[], char target[]);
int getPriority(char);
int main()
{
    char infix[100], postfix[100];
    printf("\n Enter any infix expression : ");
    gets(infix);
    strcpy(postfix, "");
    InfixtoPostfix(infix, postfix);
    printf("\n The corresponding postfix expression is : ");
    puts(postfix);
    getch();
    return 0;
}
```

Contd…

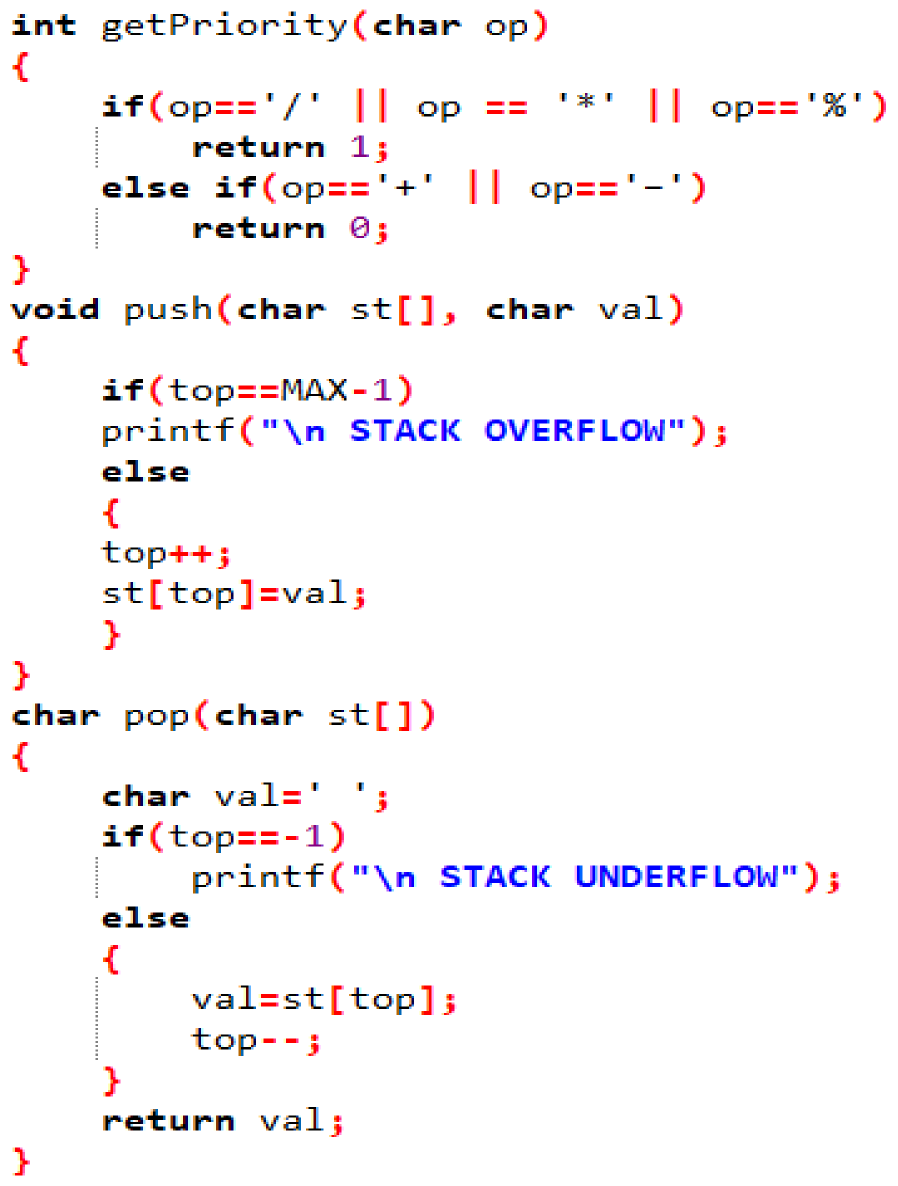
```
void InfixtoPostfix(char source[], char target[])
{
    int i=0, j=0;
    char temp;
    strcpy(target, "");
    while(source[i]!='\0')
    {
        if(source[i]=='(')
        {
            push(st, source[i]);
            i++;
        }
        else if(source[i] == ')')
        {
            while((top!=-1) && (st[top]!='('))
            {
                target[j] = pop(st);
                j++;
            }
            if(top==-1)
            {
                printf("\n INCORRECT EXPRESSION");
                exit(1);
            }
            temp = pop(st);//remove left parenthesis
            i++;
        }
```

Contd…



```
        else if(isdigit(source[i]) || isalpha(source[i]))
        {
            target[j] = source[i];
            j++;
            i++;
        }
        else if (source[i] == '+' || source[i] == '-' || source[i] == '*' || source[i] == '/' || source[i] == '%')
        {
            while( (top!=-1) && (st[top]!= '(') && (getPriority(st[top]) > getPriority(source[i])))
            {
                target[j] = pop(st);
                j++;
            }
            push(st, source[i]);
            i++;
        }
        else
        {
            printf("\n INCORRECT ELEMENT IN EXPRESSION");
            exit(1);
        }
    }
    while((top!=-1) && (st[top]!='('))
    {
        target[j] = pop(st);
        j++;
    }
        target[j]='\0';
}
```

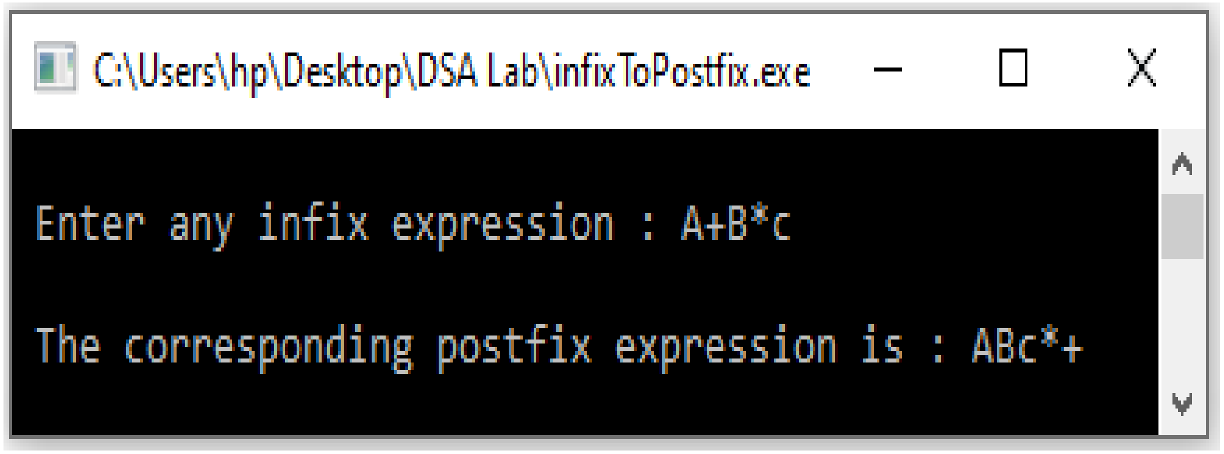
Contd…



```
int getPriority(char op)
{
    if(op=='/' || op == '*' || op=='%')
        return 1;
    else if(op=='+' || op=='-')
        return 0;
}
void push(char st[], char val)
{
    if(top==MAX-1)
    printf("\n STACK OVERFLOW");
    else
    {
    top++;
    st[top]=val;
    }
}
char pop(char st[])
{
    char val=' ';
    if(top==-1)
        printf("\n STACK UNDERFLOW");
    else
    {
        val=st[top];
        top--;
    }
    return val;
}
```

Contd…

- **Sample run and output:**



```
Enter any infix expression : A+B*c

The corresponding postfix expression is : ABc*+
```

Evaluation of postfix expressions

- Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack.
- However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values.
- The result is then pushed on to the stack. And so on until at the end of expression reached.
- The final evaluated value of postfix expression is the element at the top of stack.

Contd…

- **Algorithm to evaluate a postfix expression:**
- **Step1:** Add a “)” at the end of the postfix expression.
- **Step2:** Scan every character of the postfix expression and repeat steps 3 and 4 until “)” is encountered.
- **Step3:** IF an operand is encountered, push it on the stack. IF an operator O is encountered, then:
  - **a.** pop the top two elements from the stack as A and B .
  - **b.** Evaluate **B O A**, where A is the top most element and B is the element below A.
  - **c.** Push the result of evaluation on the stack.
- **Step4:** Set result equal to the topmost element of the stack.
- **Step5:** EXIT.

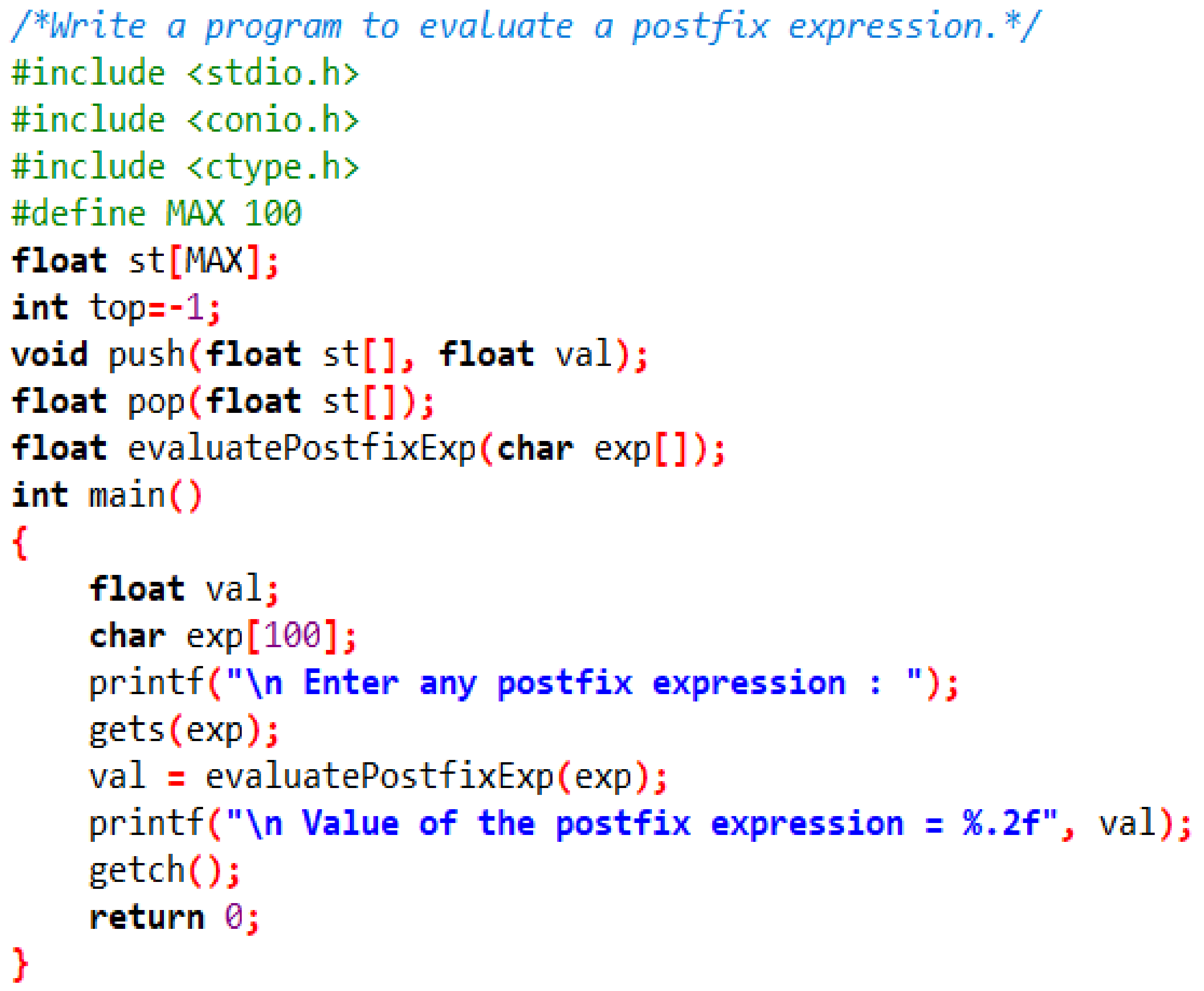
Contd…

- **Example:** Evaluate the infix expression **9 3 4 * 8 + 4 / –** using stack data structure.
- **Solution:**

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

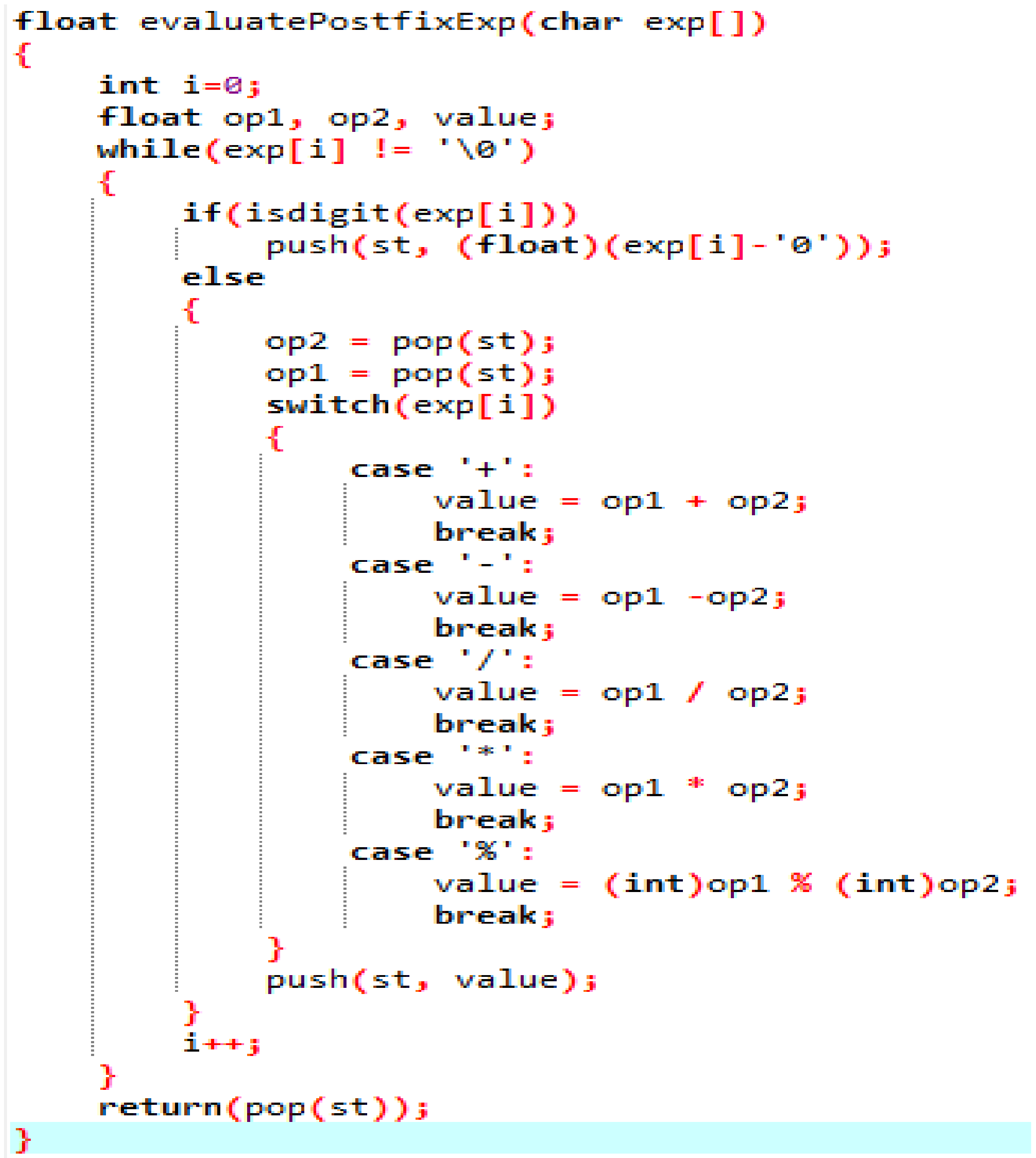
Contd…

- **Program: Write a program to evaluate a postfix expression.**



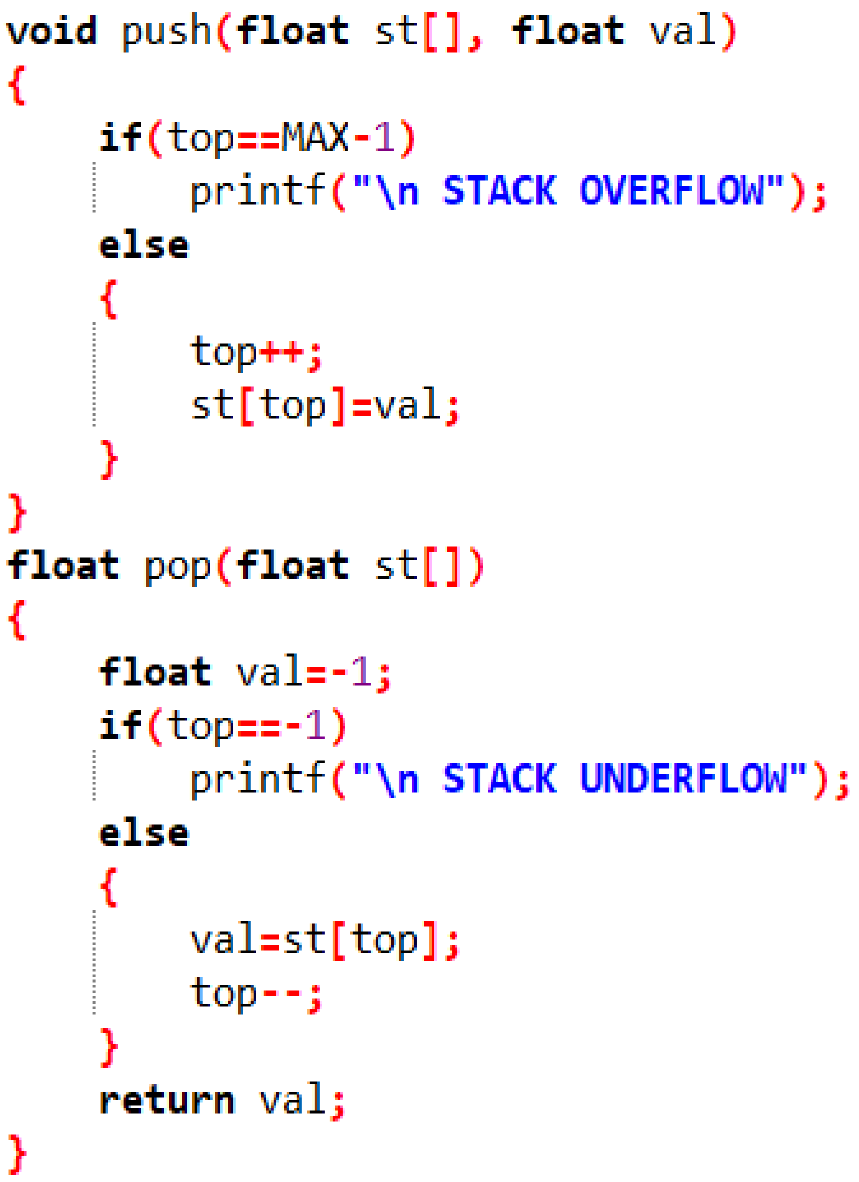
```c
/*Write a program to evaluate a postfix expression.*/
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#define MAX 100
float st[MAX];
int top=-1;
void push(float st[], float val);
float pop(float st[]);
float evaluatePostfixExp(char exp[]);
int main()
{
    float val;
    char exp[100];
    printf("\n Enter any postfix expression : ");
    gets(exp);
    val = evaluatePostfixExp(exp);
    printf("\n Value of the postfix expression = %.2f", val);
    getch();
    return 0;
}
```

Contd…



```
float evaluatePostfixExp(char exp[])
{
    int i=0;
    float op1, op2, value;
    while(exp[i] != '\0')
    {
        if(isdigit(exp[i]))
            push(st, (float)(exp[i]-'0'));
        else
        {
            op2 = pop(st);
            op1 = pop(st);
            switch(exp[i])
            {
                case '+':
                    value = op1 + op2;
                    break;
                case '-':
                    value = op1 -op2;
                    break;
                case '/':
                    value = op1 / op2;
                    break;
                case '*':
                    value = op1 * op2;
                    break;
                case '%':
                    value = (int)op1 % (int)op2;
                    break;
            }
            push(st, value);
        }
        i++;
    }
    return(pop(st));
}
```

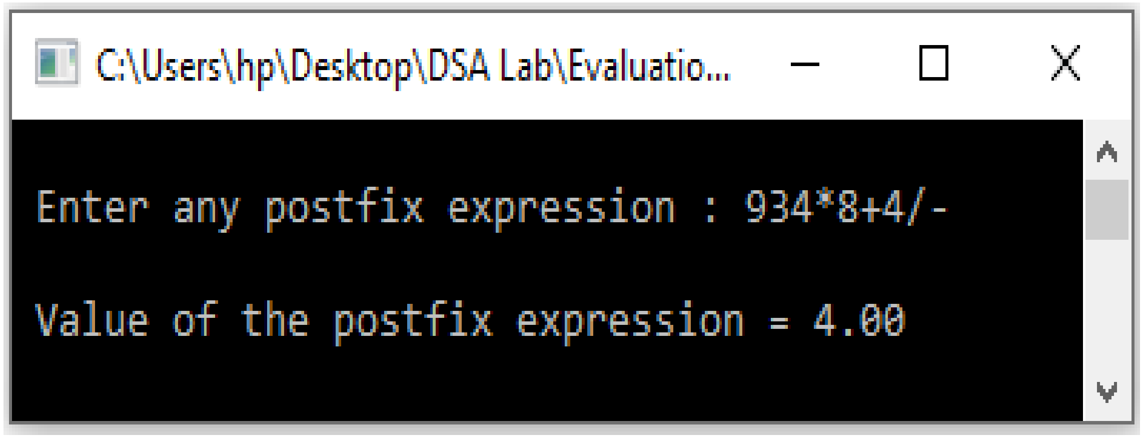
Contd…



```
void push(float st[], float val)
{
    if(top==MAX-1)
        printf("\n STACK OVERFLOW");
    else
    {
        top++;
        st[top]=val;
    }
}
float pop(float st[])
{
    float val=-1;
    if(top==-1)
        printf("\n STACK UNDERFLOW");
    else
    {
        val=st[top];
        top--;
    }
    return val;
}
```

Contd…

- **Sample Run and Output:**



```
Enter any postfix expression : 934*8+4/-

Value of the postfix expression = 4.00
```

Conversion from infix to prefix expression

- **Algorithm:**
  - **Step1:** Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.
  - **Step2:** Obtain the postfix expression of the infix expression obtained in step1.
  - **Step3:** Reverse the postfix expression to get the prefix expression.

Contd…

- **Example:** Find the corresponding prefix expression of the given infix expression: (A – B / C) * (A / K – L) .
  - **Step 1:** Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.

    (L – K / A) * (C / B – A)
  - **Step 2:** Obtain the corresponding postfix expression of the infix expression obtained as a result of Step 1.

    The expression is: (L – K / A) * (C / B – A)

    Therefore, [L – (K A /)] * [(C B /) – A]

    = [LKA/–] * [CB/A–]

    = L K A / – C B / A – *
  - **Step 3:** Reverse the postfix expression to get the prefix expression

    Therefore, the prefix expression is * – A / B C – /A K L.

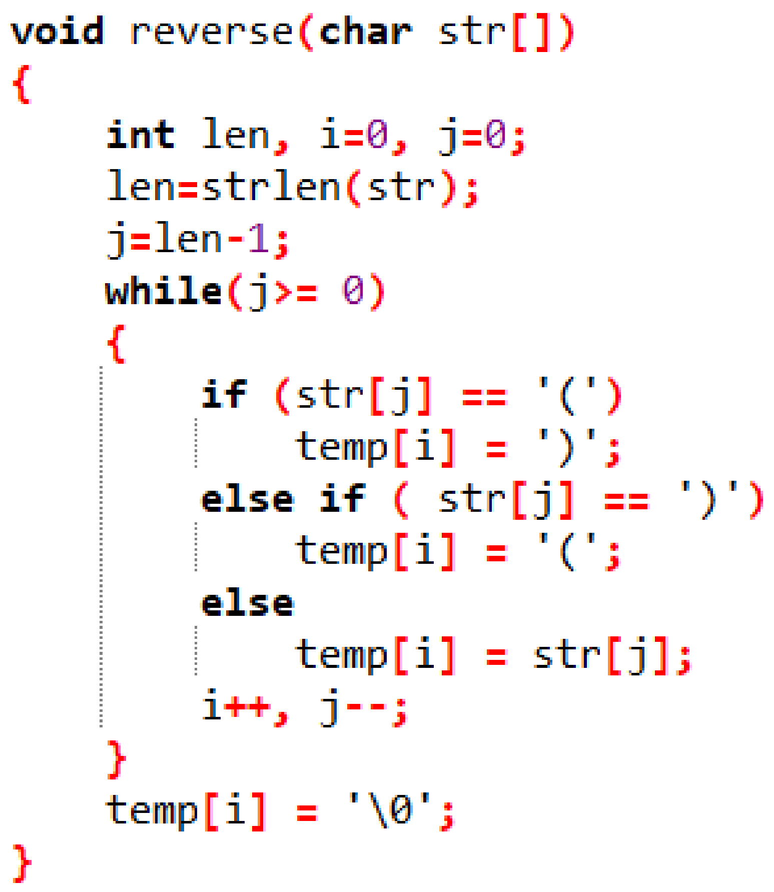
Contd…

- **Program: Write a program to convert an infix expression to a prefix expression.**

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>
#define MAX 100
char st[MAX];
int top=-1;
void reverse(char str[]);
void push(char st[], char);
char pop(char st[]);
void InfixtoPostfix(char source[], char target[]);
int getPriority(char);
char infix[100], postfix[100], temp[100];
int main()
{
    printf("Enter any infix expression : ");
    gets(infix);
    reverse(infix);
    strcpy(postfix, "");
    InfixtoPostfix(temp, postfix);
    printf("\nStep2:The corresponding postfix expression is:");
    puts(postfix);
    strcpy(temp,"");
    reverse(postfix);
    printf("\nStep3:The prefix expression is:");
    puts(temp);
    getch();
    return 0;
}
```

Contd…

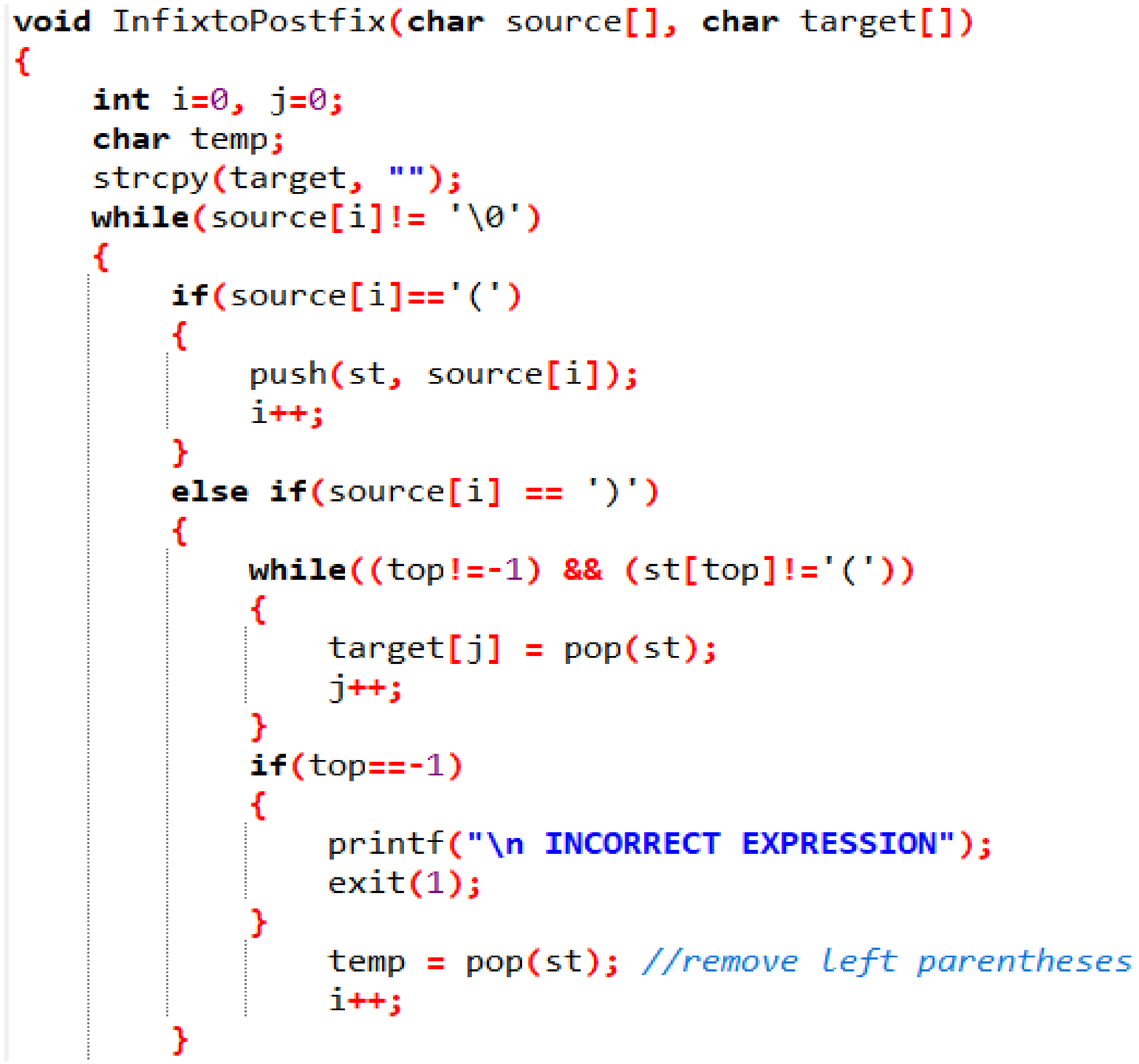
- **Continue…**



```c
void reverse(char str[])
{
    int len, i=0, j=0;
    len=strlen(str);
    j=len-1;
    while(j>= 0)
    {
        if (str[j] == '(')
            temp[i] = ')';
        else if ( str[j] == ')')
            temp[i] = '(';
        else
            temp[i] = str[j];
        i++, j--;
    }
    temp[i] = '\0';
}
```

Contd…

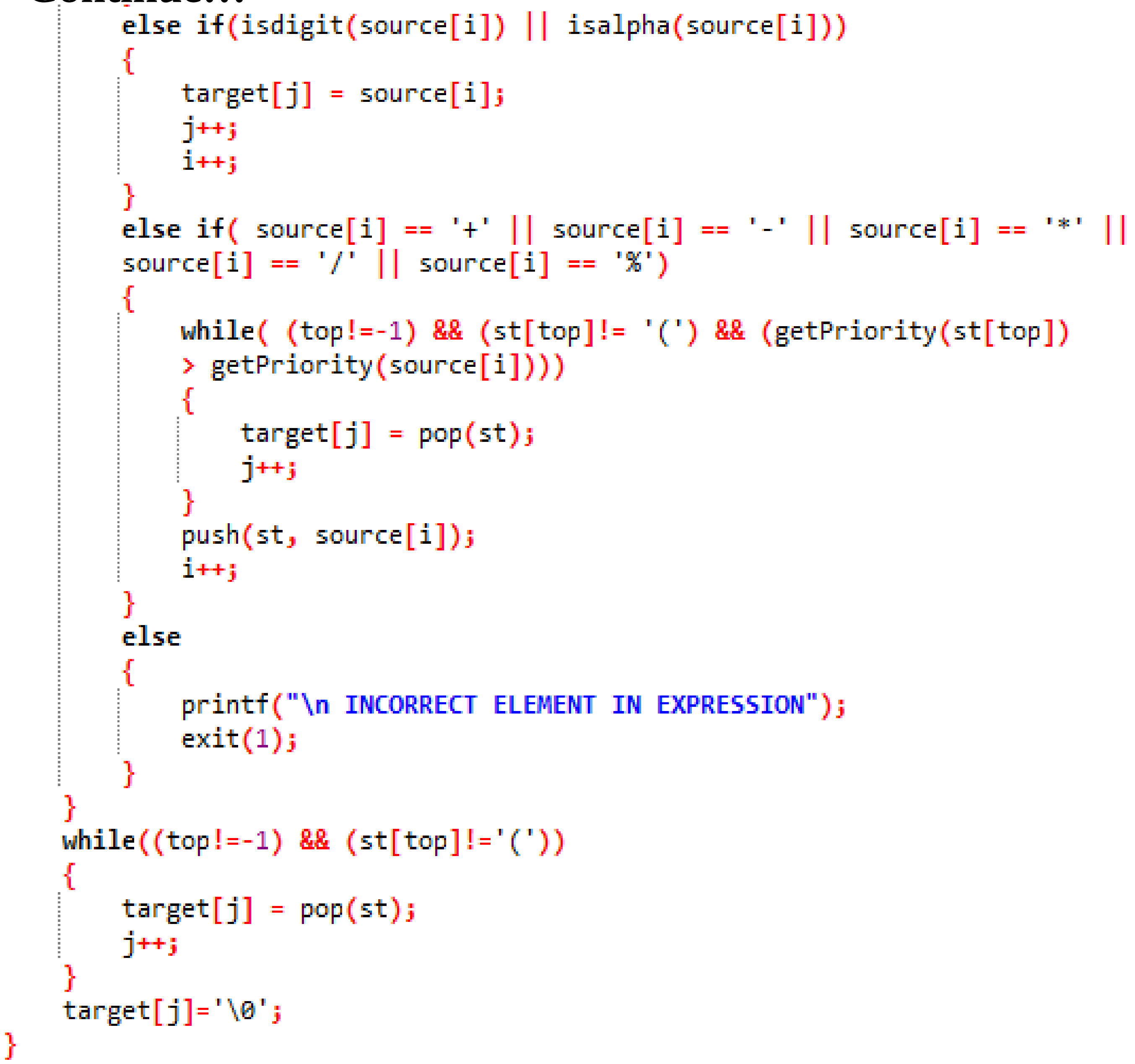
- **Continue…**



```
void InfixtoPostfix(char source[], char target[])
{
    int i=0, j=0;
    char temp;
    strcpy(target, "");
    while(source[i]!= '\0')
    {
        if(source[i]=='(')
        {
            push(st, source[i]);
            i++;
        }
        else if(source[i] == ')')
        {
            while((top!=-1) && (st[top]!='('))
            {
                target[j] = pop(st);
                j++;
            }
            if(top==-1)
            {
                printf("\n INCORRECT EXPRESSION");
                exit(1);
            }
            temp = pop(st); //remove left parentheses
            i++;
        }
```

Contd…

- **Continue…**



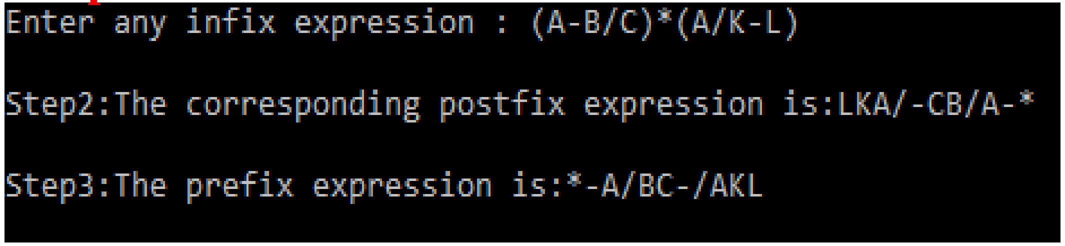
```
        else if(isdigit(source[i]) || isalpha(source[i]))
        {
            target[j] = source[i];
            j++;
            i++;
        }
        else if( source[i] == '+' || source[i] == '-' || source[i] == '*' ||
        source[i] == '/' || source[i] == '%')
        {
            while( (top!=-1) && (st[top]!= '(') && (getPriority(st[top])
            > getPriority(source[i])))
            {
                target[j] = pop(st);
                j++;
            }
            push(st, source[i]);
            i++;
        }
        else
        {
            printf("\n INCORRECT ELEMENT IN EXPRESSION");
            exit(1);
        }
    }
    while((top!=-1) && (st[top]!='('))
    {
        target[j] = pop(st);
        j++;
    }
    target[j]='\0';
}
```

Contd…

- **Continue…**

```c
int getPriority( char op)
{
    if(op=='/' || op == '*' || op=='%')
        return 1;
    else if(op=='+' || op=='-')
    return 0;
}
void push(char st[], char val)
{
    if(top==MAX-1)
        printf("\n STACK OVERFLOW");
    else
    {
        top++;
        st[top] = val;
    }
}
char pop(char st[])
{
    char val=' ';
    if(top==-1)
        printf("\n STACK UNDERFLOW");
    else
    {
        val=st[top];
        top--;
    }
    return val;
}
```

Output:



```
Enter any infix expression : (A-B/C)*(A/K-L)

Step2:The corresponding postfix expression is:LKA/-CB/A-*

Step3:The prefix expression is:*-A/BC-/AKL
```

Evaluation of prefix expressions

- **Algorithm:**
  - **Step1:** Accept the prefix expression
  - **Step2:** Repeat until all the characters in the prefix expression have been scanned.
    - **a)** Scan the prefix expression from right, one character at a time.
    - **b)** If the scanned character is an operand, push it on the operand stack.
    - **c)** If the scanned character is an operator, then
      - **i)** pop two values from the operand stack.
      - **ii)** apply the operator on the popped operands.
      - **iii)** push the result on the operand stack.
  - **Step3:** END

Contd…

- **Example:** Evaluate the following prefix expression + – 2 7 * 8 /3 9.
- **Solution:**

Character scanned	Operand stack
9	9
3	9, 3
/	3
8	3, 8
*	24
7	24 , 7
2	24, 7, 2
-	24, 5
+	29

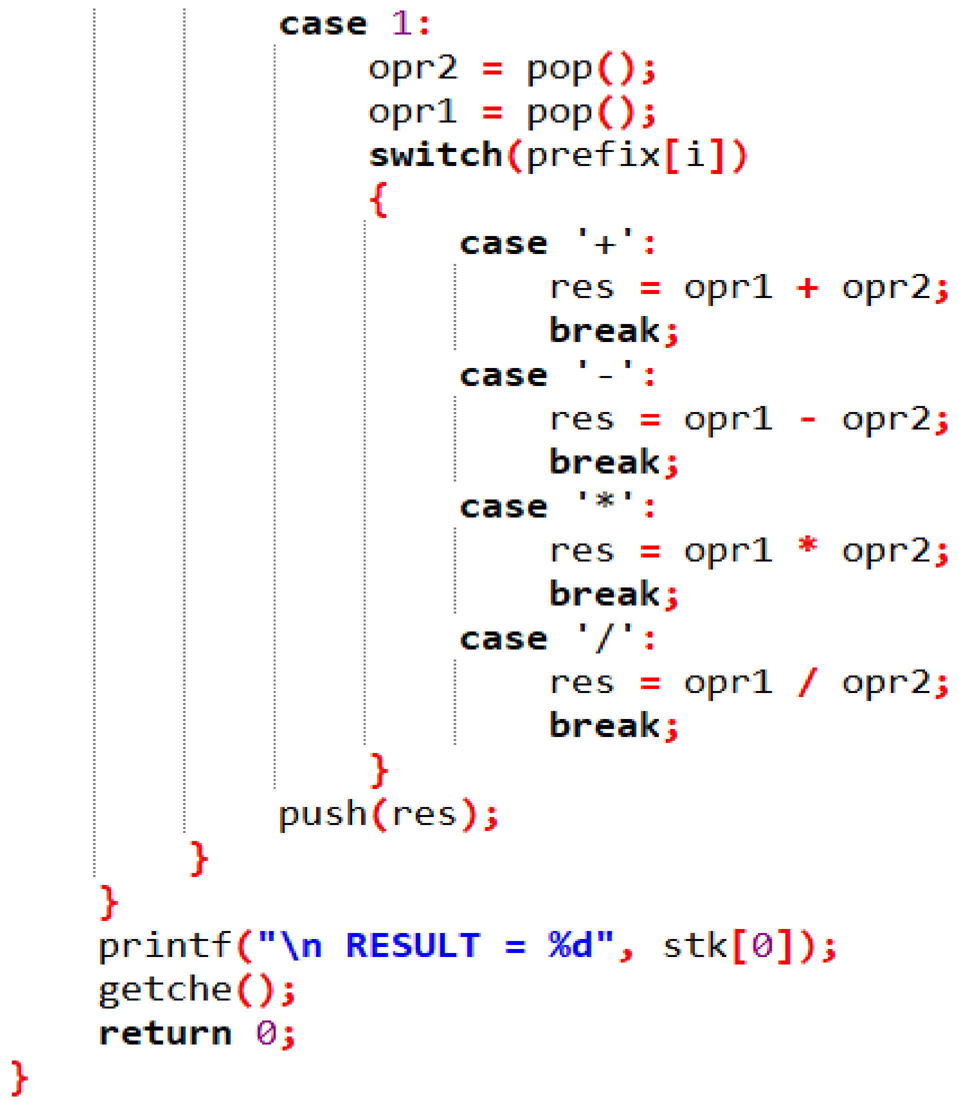
Contd…

- **Program: Write a program to evaluate a prefix expression.**

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int stk[10];
int top=-1;
int pop();
void push(int);
int get_type(char c);
int main()
{
    char prefix[10];
    int len, val, i, opr1, opr2, res;
    printf("\n Enter the prefix expression : ");
    gets(prefix);
    len = strlen(prefix);
    for(i=len-1;i>=0;i--)
    {
        switch(get_type(prefix[i]))
        {
            case 0:
                val = prefix[i] - '0';
                push(val);
                break;
```

Contd…

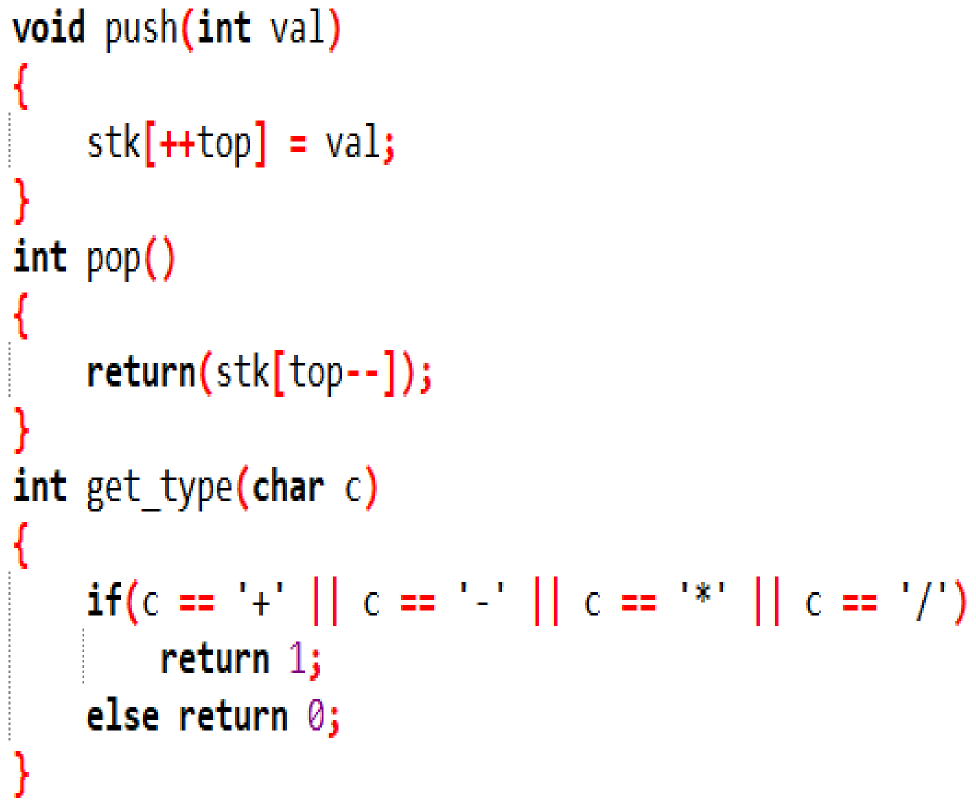
- **Continue…**



```
            case 1:
                opr2 = pop();
                opr1 = pop();
                switch(prefix[i])
                {
                    case '+':
                        res = opr1 + opr2;
                        break;
                    case '-':
                        res = opr1 - opr2;
                        break;
                    case '*':
                        res = opr1 * opr2;
                        break;
                    case '/':
                        res = opr1 / opr2;
                        break;
                }
            push(res);
        }
    }
    printf("\n RESULT = %d", stk[0]);
    getche();
    return 0;
}
```

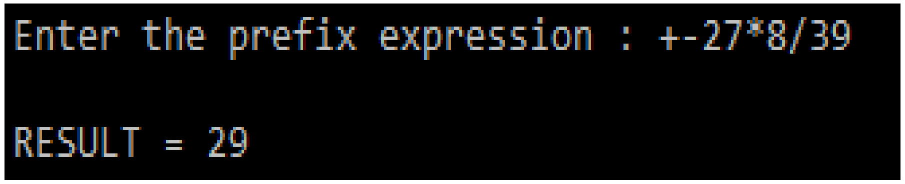
Contd…

- **Continue…**



```
void push(int val)
{
    stk[++top] = val;
}
int pop()
{
    return(stk[top--]);
}
int get_type(char c)
{
    if(c == '+' || c == '-' || c == '*' || c == '/')
        return 1;
    else return 0;
}
```

- **Output:**



```
Enter the prefix expression : +-27*8/39

RESULT = 29
```

Homework

- What is stack? Explain the various operations that can be performed on a stack.
- What is stack? Explain stack as ADT.
- What is stack? How is it different form array? Write a menu program to demonstrate the simulation of stack operations in array implementation.
- What is bracket matching problem? How it can be solved with the help of stack data structure? Explain with suitable example.
- Explain the algorithms for infix to postfix conversion and evaluation of postfix expression. Trace the algorithms with suitable example.
- Explain Infix to postfix conversion algorithm. Illustrate it with an example. What changes should be made for converting postfix to prefix.

Contd…

- Trace out infix to postfix conversion algorithm with given infix expression A+(((B-C)*(D-E)+F)/g)+(H-I). Evaluate the postfix expression acquired from above for the given values: A=6, B=2, C=4, D=3, E=8, F= 2, G=3, H=5 , and I=1.
- What is postfix expression? How can you use stack to convert an infix expression to postfix? Convert infix expression (A+B)*(C-D) into postfix expression.
- What is prefix expression? How can you use stack to convert an infix expression to prefix? Convert infix expression (A+B)*(C-D) into prefix expression.

Lab 2

- Write a program to create and perform push, pop, peek and display operation on a stack with integer data.
- Write a program to create and perform push, pop, peek and display operation on a stack with character data.
- Write a program to reverse a list of given numbers.
- Write a program to check nesting of parentheses using a stack.
- Write a program to convert an infix expression into its equivalent postfix notation.
- Write a program to evaluate a postfix expression.
- Write a program to convert an infix expression to a prefix expression.
- Write a program to evaluate a prefix expression.

Thank You !