STACK APPLICATIONS:

1) REVERSAL OF STRING USING STACK:

Reverse String using STACK in C - This program will **read a string and reverse the string using Stack push and pop operations** in C programming Language.

Reversing string is an operation of Stack by using Stack we can reverse any string, here we implemented a program in C - this will reverse given string using Stack.

The logic behind to implement this program:

1. Read a string.
2. **Push all characters until NULL is not found** - Characters will be stored in stack variable.
3. **Pop all characters until NULL is not found** - As we know stack is a **LIFO** technique, so last character will be pushed first and finally we will get reversed string in a variable in which we store inputted string.

```
/* Program of reversing a string using stack */

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#define MAX 20

int top = -1;

char stack[MAX];

char pop();

void push(char);

main()

{

        char str[20];

        unsigned int i;

        printf("Enter the string : " );

        gets(str);
```

```
        /*Push characters of the string str on the stack */
        for(i=0;i<strlen(str);i++)
                push(str[i]);
        /*Pop characters from the stack and store in string str */
        for(i=0;i<strlen(str);i++)
                str[i]=pop();
        printf("Reversed string is : ");
        puts(str);
}/*End of main()*/
void push(char item)
{
        if(top == (MAX-1))
        {
                printf("Stack Overflow\n");
                return;
        }
        stack[++top] =item;
}/*End of push()*/
char pop()
{
        if(top == -1)
        {
                printf("Stack Underflow\n");
                exit(1);
        }
```

```
        return stack[top--];

}/*End of pop()*/
```

OUTPUT:

Input a string: Hello World!

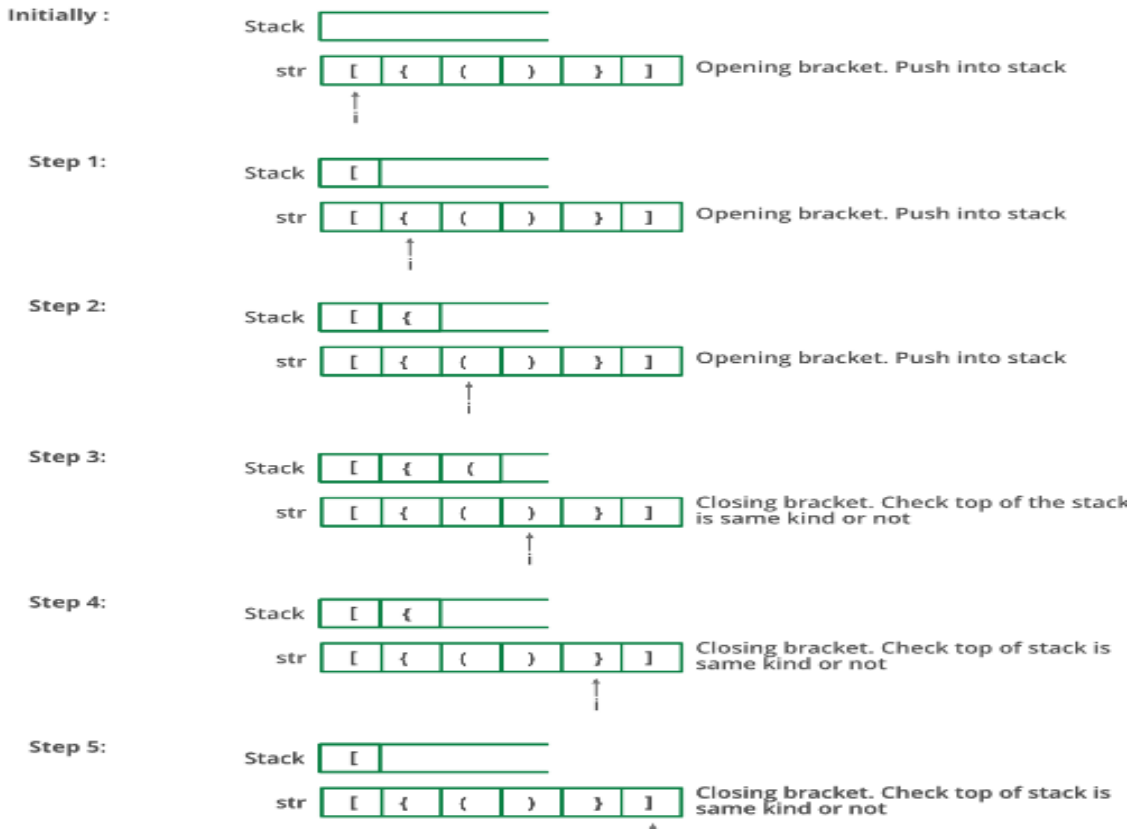
Reversed String is: !dlroW olleH

2) CHECK THE VALIDITY OF AN EXPRESSION:

Algorithm:

- Declare a character stack S.
- Now traverse the expression string exp.
    1. If the current character is a starting bracket (**'(' or '{' or '['**) then push it to stack.
    2. If the current character is a closing bracket (**')' or '}' or ']'**) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
- After complete traversal, if there is some starting bracket left in stack then "not balanced"

Below image is a dry run of the above approach:



```
/*Program to check nesting of parentheses using stack*/
#include<stdio.h>
#define MAX 30
int top=-1;
int stack[MAX];
void push(char);
char pop();
int match(char a,char b);
main()
{
        char exp[MAX];
        int valid;
        printf("Enter an algebraic expression : ");
        gets(exp);
        valid=check(exp);
        if(valid==1)
                printf("Valid expression\n");
        else
                printf("Invalid expression\n");
}
int check(char exp[] )
{
        int i;
        char temp;
        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)    /*stack empty*/
                {
                        printf("Right parentheses are more than left parentheses\n");
                        return 0;
                }
                else
                {
                        temp=pop();
                        if(!match(temp, exp[i]))
                        {
                                printf("Mismatched parentheses are : ");
                                printf("%c and %c\n",temp,exp[i]);
                                return 0;
                        }
                }
}
if(top==-1) /*stack empty*/
{
        printf("Balanced Parentheses\n");
        return 1;
}
```

```
        else
        {
                printf("Left parentheses more than right parentheses\n");
                return 0;
        }
}/*End of main()*/
int match(char a,char b)
{
        if(a=='[' && b==']')
                return 1;
        if(a=='{' && b=='}')
                return 1;
        if(a=='(' && b==')')
                return 1;
        return 0;
}/*End of match()*/

void push(char item)
{
        if(top==(MAX-1))
        {
                printf("Stack Overflow\n");
                return;
        }
        top=top+1;
```

```
        stack[top]=item;
}/*End of push()*/
char pop()
{
        if(top==-1)
        {
                printf("Stack Underflow\n");
                exit(1);
        }
        return(stack[top--]);
}/*End of pop()*/
```

OUTPUT:

1)

```
Enter an algebraic expression : {[()]}
Balanced Parentheses
Valid expression
```

2)

```
Enter an algebraic expression : {[(}]
Mismatched parentheses are : ( and }
Invalid expression
```

3) FUNCTION CALLS:

4.6.3 Function calls

The function calls behave in LIFO manner, i.e. the function that is called first is the last one to finish execution or we can say that the function calls return in the reverse order of their invocation. So stack is the perfect data structure to implement function calls.

A stack is maintained during the execution of the program called program stack or run-time stack. Whenever a function is called the program stores all the information associated with this call in a structure called activation record and this activation record is pushed on the run-time stack. Activation record also known as stack frame consists of the following data-

(i) Parameters and local variables.
(ii) Pointer to previous activation record i.e. activation record of the caller.
(iii) Return address i.e. the instruction in the caller function which is immediately after the function call.
(iv) Return value if function is not void.

The function whose activation record is at the top of the stack is the one that is currently being executed. Whenever a function is called, its activation record is pushed on the stack. When a function terminates its activation record is popped from the stack and after this the activation record of its caller comes at the top. Let us take a C program example in which `main()` calls `f1()`, `f1()` calls `f2()` and `f2()` calls `f3()`. The different stages of run-time stack in this case are shown in figure 4.15. AR in the figure denotes activation record.

If a function calls itself, then the program creates different activation records for different calls of the same function(recursive calls). So we can see that there is no need of any special procedure to implement recursion.

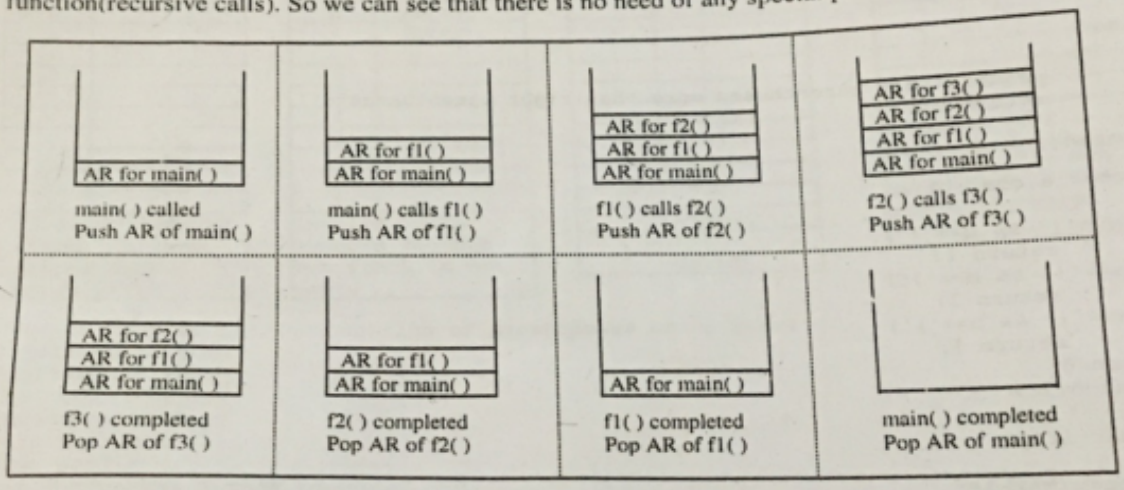


Figure 4.15

4) CONVERSION and EVALUATION OF INFIX TO POSTFIX FORM USING STACK:

Infix expression: The expression of the form a op b. When an operator is in-between every pair of operands.

Postfix expression: The expression of the form a b op. When an operator is followed for every pair of operands.

Why postfix representation of the expression?
The compiler scans the expression either from left to right or from right to left.

Consider the below expression: a op1 b op2 c op3 d

If op1 = +, op2 = *, op3 = +

The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it. The result is then added to d after another scan.

The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.

The corresponding expression in postfix form is: abc*+d+. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.

Algorithm
1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
…..**3.1** If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
…..**3.2** Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

```
/*Program for conversion of infix to postfix and evaluation of postfix.

It will evaluate only single digit numbers*/

#include<stdio.h>

#include<string.h>

#include<math.h>

#include<stdlib.h>

#define BLANK ' '

#define TAB '\t'

#define MAX 50

void push(long int symbol);
```

```
long int pop();
void infix_to_postfix();
long int eval_post();
int priority(char symbol);
int isEmpty();
int white_space(char);
char infix[MAX], postfix[MAX];
long int stack[MAX];
int top;
int main()
{
        long int value;
        top=-1;
        printf("Enter infix : ");
        gets(infix);
        infix_to_postfix();
        printf("Postfix : %s\n",postfix);
        value=eval_post();
        printf("Value of expression : %ld\n",value);
}/*End of main()*/
void infix_to_postfix()
{
        unsigned int i,p=0;
        char next;
        char symbol;
```

```
for(i=0;i<strlen(infix);i++)
{
        symbol=infix[i];
        if(!white_space(symbol))
        {
                switch(symbol)
                {
                case '(':
                        push(symbol);
                        break;
                case ')':
                        while((next=pop())!='(')
                                postfix[p++] = next;
                        break;
                case '+':
                case '-':
                case '*':
                case '/':
                case '%':
                case '^':
                        while( !isEmpty( ) &&  priority(stack[top])>= priority(symbol) )
                                postfix[p++]=pop();
                        push(symbol);
                        break;
                default: /*if an operand comes*/
```

```
                        postfix[p++]=symbol;
                    }
                }
        }
        while(!isEmpty( ))
                postfix[p++]=pop();
        postfix[p]='\0'; /*End postfix with'\0' to make it a string*/
}/*End of infix_to_postfix()*/
/*This function returns the priority of the operator*/
int priority(char symbol)
{
        switch(symbol)
        {
        case '(':
                return 0;
        case '+':
        case '-':
                return 1;
        case '*':
        case '/':
        case '%':
                return 2;
        case '^':
                return 3;
        default :
```

```
                return 0;
        }
}/*End of priority()*/
void push(long int symbol)
{
        if(top>MAX)
        {
                printf("Stack overflow\n");
                exit(1);
        }
        stack[++top]=symbol;
}/*End of push()*/
long int pop()
{
        if( isEmpty() )
        {
                printf("Stack underflow\n");
                exit(1);
        }
        return (stack[top--]);
}/*End of pop()*/
int isEmpty()
{
        if(top==-1)
                return 1;
```

```
        else
                return 0;
}/*End of isEmpty()*/
int white_space(char symbol)
{
        if( symbol == BLANK || symbol == TAB )
                return 1;
        else
                return 0;
}/*End of white_space()*/

long int eval_post()
{
        long int a,b,temp,result;
        unsigned int i;

        for(i=0;i<strlen(postfix);i++)
        {
                if(postfix[i]<='9' && postfix[i]>='0')
                        push(postfix[i]-'0');
                else
                {
                        a=pop();
                        b=pop();
                        switch(postfix[i])
```

```
                {
                case '+':
                        temp=b+a; break;
                case '-':
                        temp=b-a;break;
                case '*':
                        temp=b*a;break;
                case '/':
                        temp=b/a;break;
                case '%':
                        temp=b%a;break;
                case '^':
                        temp=pow(b,a);
                }
                push(temp);
            }
        }
        result=pop();
        return result;
}/*End of eval_post */
```

OUTPUT:1)

```
Enter infix: 3+5*(74)^2

Postfix: 35742^*+

Value of expression : 48
```

```
Enter infix : 7+5*3^2/(92^2)+6*4

Postfix: 7532^*922^/+64*+

Value of expression : 40
```