**Stack**

#include <iostream>

using namespace std;

int stack[10],n=10,top=-1;

void push(int value)

{

if(top>=n-1)

cout<<"Stack Overflow"<<endl;

else

{

top++;

stack[top]=value;

}

}

void pop()

{

if(top<=-1)

cout<<"Stack Underflow"<<endl;

else

{

cout<<"The popped element is "<< stack[top] <<endl;

top--;

}

}

void display()

{

if(top>=0)

{

cout<<"Stack elements are:"<<endl;

for(int i=top; i>=0; i--)

{

cout<<stack[i]<<" ";

cout<<endl;

}

}

else

{

cout<<"Stack is empty";

}

}

int main()

{

int choice, value;

cout<<"1) Push in stack"<<endl;

cout<<"2) Pop from stack"<<endl;

cout<<"3) Display stack"<<endl;

cout<<"4) Exit"<<endl;

do

{

cout<<"Enter choice: ";

cin>>choice;

switch(choice)

{

case 1:

{

cout<<"Enter value to be pushed:";

cin>>value;

push(value);

break;

}

case 2:

{

pop();

break;

}

case 3:

{

display();

break;

}

case 4:

{

cout<<"Exit"<<endl;

break;

}

default:

{

cout<<"Invalid Choice"<<endl;

}

}

}while(choice!=4);

return 0;

}

**Stack Recursion (Tower of Hanoi)**

#include<iostream>

using namespace std;

void towerOfHanoi(int n, char from\_rod, char to\_rod,char aux\_rod)

{

if (n == 0)

{

return;

}

towerOfHanoi(n - 1, from\_rod, aux\_rod, to\_rod);

cout << "Move disk " << n << " from rod " << from\_rod<< " to rod " << to\_rod << endl;

towerOfHanoi(n - 1, aux\_rod, to\_rod, from\_rod);

}

int main()

{

int N=2;

towerOfHanoi(N, 'A', 'C', 'B');

return 0;

}

**Infix to Postfix Conversion**

#include<iostream>

#include<stdlib.h>

#include<math.h>

#include <string.h>

using namespace std;

char infix\_string[20], postfix\_string[20];

int top;

int stack[20];

int pop();

int precedence(char symbol);

int isEmpty();

void infix\_to\_postfix();

void push(long int symbol);

int main()

{

top = -1;

cout<<"\nINPUT THE INFIX EXPRESSION : ";

cin>>infix\_string;

infix\_to\_postfix();

cout<<"\nEQUIVALENT POSTFIX EXPRESSION : "<<postfix\_string;

return 0;

}

void infix\_to\_postfix()

{

int count, temp = 0;

char next;

char symbol;

for(count = 0; count < strlen(infix\_string); count++)

{

symbol = infix\_string[count];

{

switch(symbol)

{

case '(':

push(symbol);

break;

case ')':

while((next = pop()) != '(')

{

postfix\_string[temp++] = next;

}

break;

case '+':

case '-':

case '\*':

case '/':

case '^':

while(!isEmpty() && precedence(stack[top]) >= precedence(symbol)) // Check precedence and push the higher one

postfix\_string[temp++] = pop();

push(symbol);

break;

default:

postfix\_string[temp++] = symbol;

}

}

}

while(!isEmpty())

{

postfix\_string[temp++] = pop();

}

postfix\_string[temp] = '\0';

}

int precedence(char symbol)

{

switch(symbol)

{

case '(':

return 0;

case '+':

case '-':

return 1;

case '\*':

case '/':

case '%':

return 2;

case '^':

return 3;

default:

return 0;

}

}

void push(long int symbol)

{

if(top > 20)

{

cout<<"Stack Overflow\n";

exit(1);

}

top = top + 1;

stack[top] = symbol;

}

int isEmpty()

{

if(top == -1)

{

return 1;

}

else

{

return 0;

}

}

int pop()

{

if(isEmpty())

{

printf("Stack is Empty\n");

exit(1);

}

return(stack[top--]);

}

**Infix to Prefix Conversion**

#include<iostream>

#include<string.h>

//#include<stdlib.h>

#define MAX 20

using namespace std;

void push(int);

char pop();

void infix\_to\_prefix();

int precedence (char);

char stack[20],infix[20],prefix[20];

int top = -1;

int main()

{

cout<<"\nINPUT THE INFIX EXPRESSION : ";

cin>>infix;

infix\_to\_prefix();

cout<<"EQUIVALENT PREFIX NOTATION : "<<prefix;

return 0;

}

void push(int pos)

{

if(top == MAX-1)

{

cout<<"\nSTACK OVERFLOW\n";

}

else

{

top++;

stack[top] = infix[pos];

}

}

char pop()

{

char ch;

if(top < 0)

{

cout<<"\nSTACK UNDERFLOW\n";

exit(0);

}

else

{

ch = stack[top];

stack[top] = '\0';

top--;

return(ch);

}

return 0;

}

void infix\_to\_prefix()

{

int i = 0,j = 0;

strrev(infix);

while(infix[i] != '\0')

{

if((infix[i] >= 'a' && infix[i] <= 'z') || (infix[i] >= 'A' && infix[i] <= 'Z'))

{

prefix[j] = infix[i];

j++;

i++;

}

else if(infix[i] == ')' || infix[i] == '}' || infix[i] == ']')

{

push(i);

i++;

}

else if(infix[i] == '(' || infix[i] == '{' || infix[i] == '[')

{

if(infix[i] == '(')

{

while(stack[top] != ')')

{

prefix[j] = pop();

j++;

}

pop();

i++;

}

else if(infix[i] == '[')

{

while(stack[top] != ']')

{

prefix[j] = pop();

j++;

}

pop();

i++;

}

else if(infix[i] == '{')

{

while(stack[top] != '}')

{

prefix[j] = pop();

j++;

}

pop();

i++;

}

}

else

{

if(top == -1)

{

push(i);

i++;

}

else if( precedence(infix[i]) < precedence(stack[top]))

{

prefix[j] = pop();

j++;

while(precedence(stack[top]) > precedence(infix[i]))

{

prefix[j] = pop();

j++;

if(top < 0)

{

break;

}

}

push(i);

i++;

}

else if(precedence(infix[i]) >= precedence(stack[top]))

{

push(i);

i++;

}

}

}

while(top != -1)

{

prefix[j] = pop();

j++;

}

strrev(prefix);

prefix[j] = '\0';

}

int precedence(char alpha)

{

if(alpha == '+' || alpha =='-')

{

return(1);

}

if(alpha == '\*' || alpha =='/')

{

return(2);

}

return 0;

}