K. J. Somaiya College of Engineering, Mumbai-77
(Autonomous College Affiliated to University of Mumbai)

Batch:_____C1_____ Roll No.:16010122257

Experiment No. 3

Grade: AA / AB / BB / BC / CC / CD /DD

Title: Implementation of Basic operations on stack using Array and Linked List-Create, Insert, Delete, Peek.

Objective: To implement Basic Operations on Stack i.e. Create, Push, Pop, Peek

Expected Outcome of Experiment:

CO	Outcome
1	Explain the different data structures used in problem solving

Books/ Journals/ Websites referred:

1. *Fundamentals Of Data Structures In C* – Ellis Horowitz, Satraj Sahni, Susan Anderson-Fred
2. *An Introduction to data structures with applications* – Jean Paul Tremblay, Paul G. Sorenson
3. *Data Structures A Pseudo Approach with C* – Richard F. Gilberg & Behrouz A. Forouzan
4. *https://www.geeksforgeeks.org/stack-data-structure-introduction-program/*

Abstract:

A Stack is an ordered collection of elements , but it has a special feature that deletion and insertion of elements can be done only from one end, called the top of the stack(TOP). The order may be LIFO(Last In First Out) or FILO(First In Last Out).
Students need to first try and understand the implementation of using arrays. Once comfortable with the concept, they can further implement stacks using linked list as well.

Related Theory: -

Stack is a linear data structure which follows a particular order in which the operations are performed. It works on the mechanism of Last in First out (LIFO).

List 5 Real Life Examples:

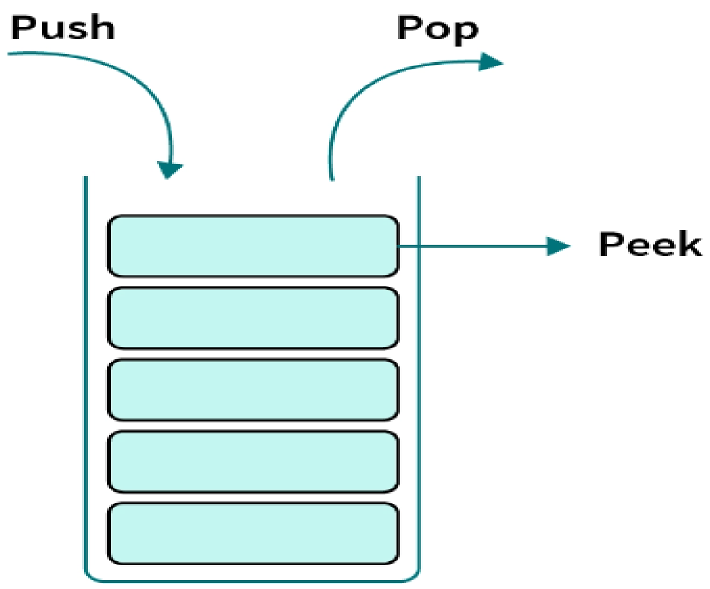
1)deck of cards,

2)piles of books, money,plates,newspapers etc.

- 3) Forward-backward surfing in the browser.
- 4) Undo/Redo button/operation in word processors.

5)Floors in a building.

Diagram:



Explain Stack ADT:

Abstract typedef StackType(ElementType
ele)
Condition: none

Stack ADT: Operator definition

1. Abstract StackType CreateStack()
Precondition: none
Postcondition: EmptyStack is created

2. Abstract StackType PushStack(StackType Stack,
ElementType Element)
Precondition: Stack not full or NotFull(Stack)= True
Postcondition: stack= stack + Element at the top
Or Stack= original stack with new Element at the top

Stack ADT: Operator definition
3. Abstract ElementType PopStack(StackType stack)
Precondition: Stack not empty or NotEmpty(Stack)= True
Postcondition: PopStack= element at the top,
Stack = stack - Element at the top
Or Stack= original stack without top Element

4. Abstract DestroyStack(StackType Stack)
Precondition: Stack not empty or NotEmpty(Stack)= True
Postcondition: Element from the stack are removed one by
one starting from top to bottom.
Empty(Stack)= True

Stack ADT: Operator definition

5. Abstract Boolean NotFull(StackType stack)
Precondition: none
Postcondition: NotFull(Stack)= true if Stack is not full
NotFull(Stack)= False if Stack is full.

6. Abstract Boolean NotEmpty(StackType stack)
Precondition: none
Postcondition: NotEmpty(Stack)= true if Stack is not empty
~Empty(Stack)= False if Stack is empty.

Stack ADT: Operator definition

7. Abstract ElementType Peep(StackType stack)
Precondition: Stack not empty or NotEmpty(Stack)= True
Postcondition: PeepStack= element at the top,
Stack = original stack

Algorithm for creation, insertion, deletion, displaying an element in stack:

Stack ADT: Array Implementation

1. Algorithm StackType CreateStack()

```
{ integer StackTop =-1;

Return stack;

}
```

2. Algorithm StackType PushStack(StackType Stack, ElementType Element)

```
{

if NotFull(Stack)= True

stack[++StackTop]= Element

Else “Error Message”

}
```

Stack ADT: Array Implementation

3. Algorithm ElementType PopStack(StackType stack)

```
{ if NotEmpty(Stack)= True
Return Stack[StackTop--]
Else print “Error Message”
}
```

```
4. Abstract DestroyStack(StackType Stack)
{ if NotEmpty(Stack) = true
while(NotEmpty(Stack))
print PopStack(Stack)
else print “Error Message”
}
```

Stack ADT: Array Implementation

```
5. Abstract Boolean NotFull(StackType stack)
{ if NotFull(Stack)
return True

else

return False

}
```

```
6. Abstract Boolean NotEmpty(StackType stack)
{ if NotEmpty(Stack)
retrun True

else

return False

}
```

Stack ADT: Array Implementation

```
7. Abstract ElementType Peek(StackType stack)
{ if NotEmpty(Stack)= True
Return Stack[StackTop]
Else print “Error Message”
}
```

Implementation Details:

Assumptions made for Input:

Valid Inputs: Assuming that valid inputs will be provided by the user throughout the execution of the program. As an example, when entering the size of the stack or choosing menu options, it's expected that the user will provide integers within a reasonable range.

Size of Stack: The user is expected to provide a positive integer value as the size of the stack. Negative values or non-numeric inputs for the stack size are not considered.

Integer Inputs: The program assumes that valid integer inputs will be provided by the user for pushing elements onto the stack and for selecting menu options.

No Error Handling: Extensive error handling isn't implemented by the program for cases where non-integer inputs are given or while performing stack operations on an empty stack.

Built-In Functions/Header Files Used: (exit() etc)

stdio.h: The program includes the standard I/O header file (<stdio.h>) for using functions like printf and scanf for input and output operations.

printf: Used for printing messages and formatted output to the console.

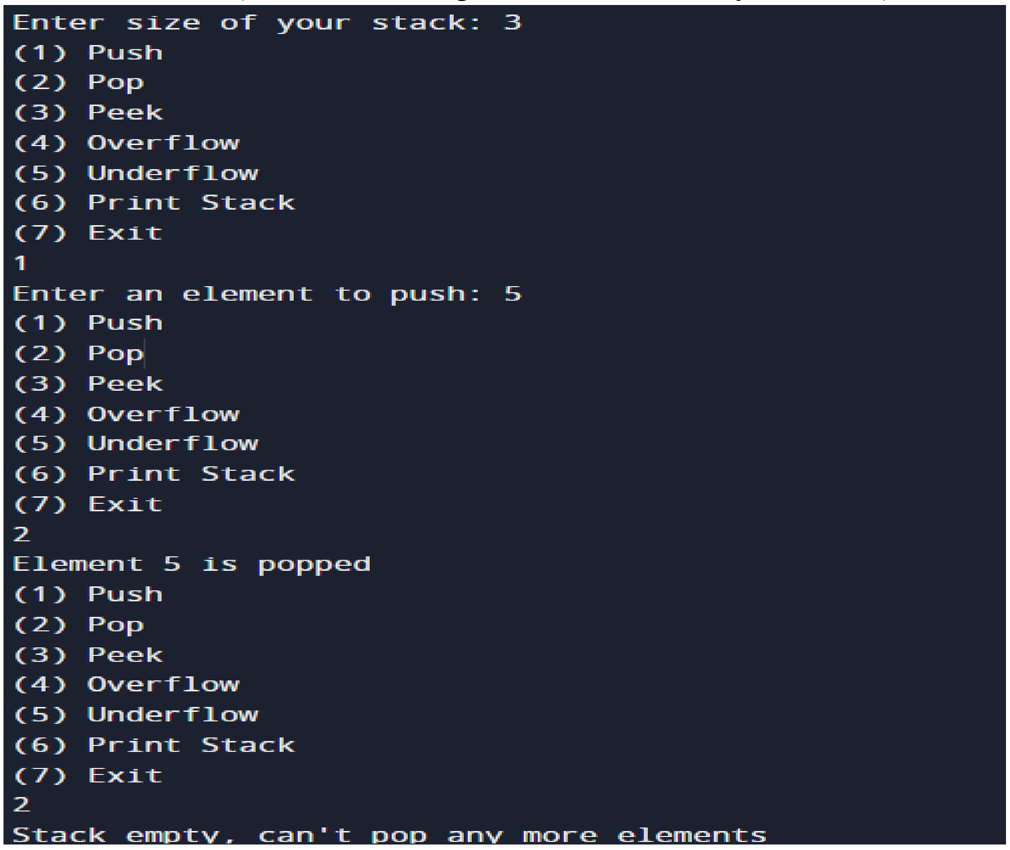
scanf: Used for reading input from the user through the console.

Program source code:

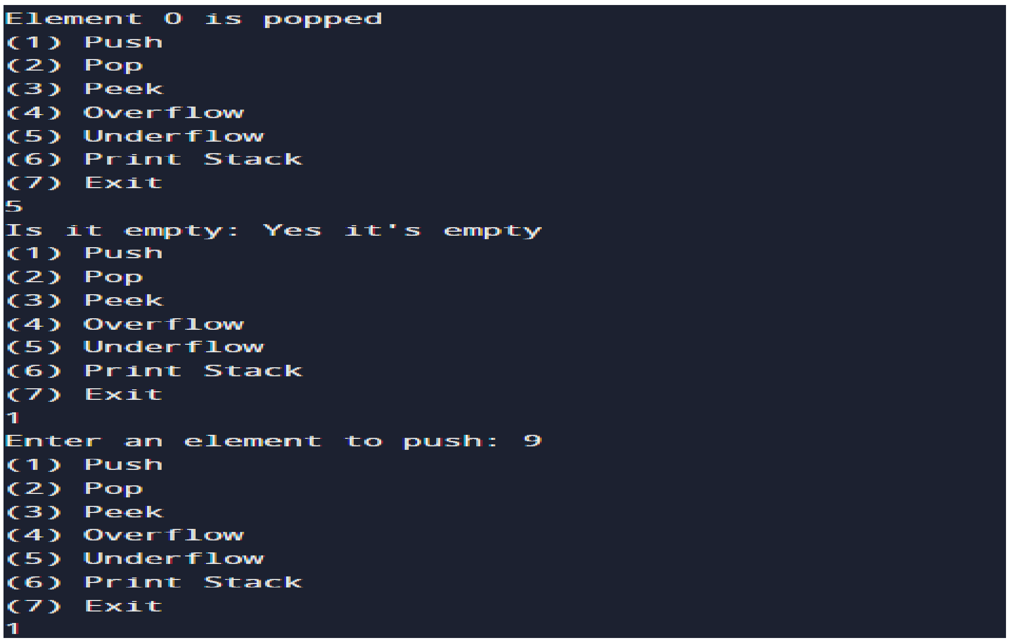
```
#include <stdio.h>

int top = -1;

void push(int *arr, int element, int m) {

    if (top >= m - 1)

        printf("Stack Full, can't push more elements\n");

    else

        arr[++top] = element;

}
```

```
int pop(int *arr) {
    if (top != -1) {
        int tempvar = arr[top];
        arr[top--] = 0;
        return tempvar;
    }
    printf("Stack empty, can't pop any more elements\n");
    return 0;
}


int peek(int *arr) {
    return arr[top];
}
char *isempty(int *arr) {
    if (top == -1)
        return "Yes it's empty";
    return "No it has elements";
}
char *overflow(int *arr, int m) {
    if (top >= m - 1)
        return "Yes, it overflows";
    return "No it doesn't overflow";
}
void printStack(int *arr) {
    if (top == -1) {
        printf("Stack is empty\n");

        return;

    }

    printf("Stack contents:\n");

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

        printf("%d\n", arr[i]);

}

int main() {

    int m;

    printf("Enter size of your stack: ");

    scanf("%d", &m);

    int stk[m];

    int j = -1;

    while (j != 7) {

        printf("(1) Push\n");

        printf("(2) Pop\n");

        printf("(3) Peek\n");

        printf("(4) Overflow\n");

        printf("(5) Underflow\n");

        printf("(6) Print Stack\n");

        printf("(7) Exit\n");

        scanf("%d", &j);

while (getchar() != '\n');

 if (j == 1) {

            int element;

            printf("Enter an element to push: ");
            scanf("%d", &element);
            push(stk, element, m);
        } else if (j == 2)
            printf("Element %d is popped\n", pop(stk));
        else if (j == 3)
            printf("Top peek Element is: %d\n", peek(stk));
        else if (j == 4)
            printf("Is overflow: %s\n", overflow(stk, m));
        else if (j == 5)
            printf("Is it empty: %s\n", isempty(stk));
        else if (j == 6)
            printStack(stk);
        else if (j == 7)
            break;
        else
            printf("Please Enter correct Option\n");
    }
    return 0;
}
```

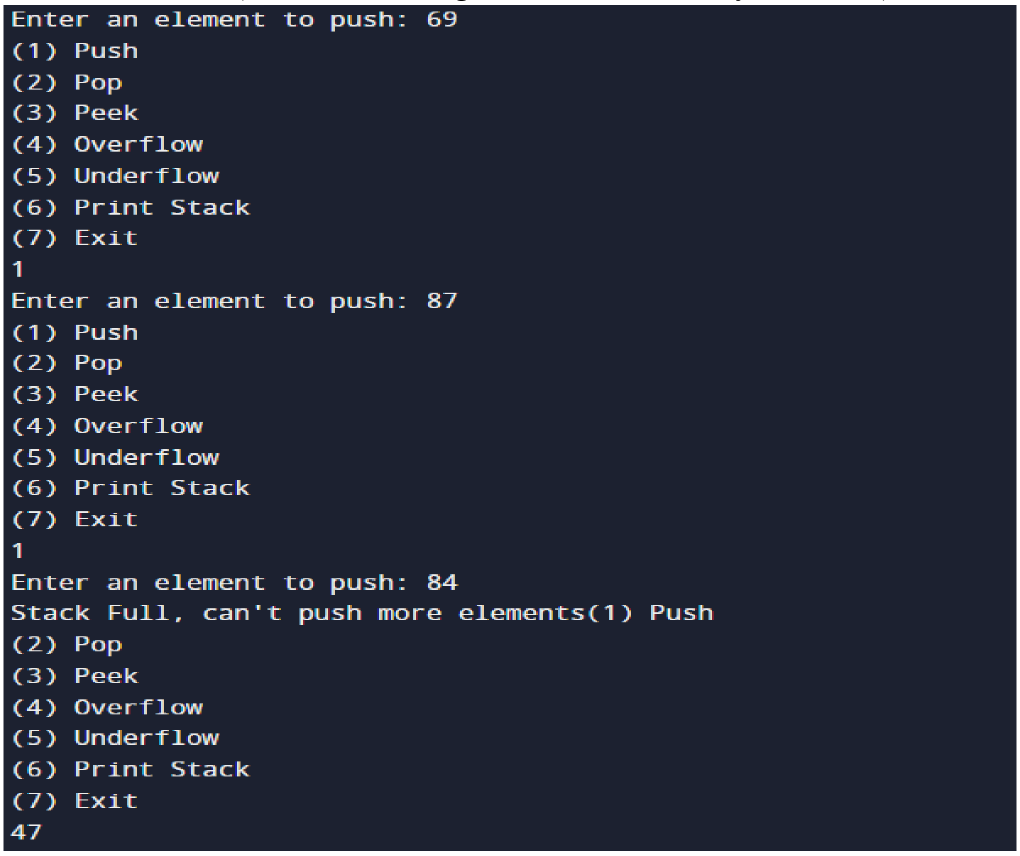
Output Screenshots:



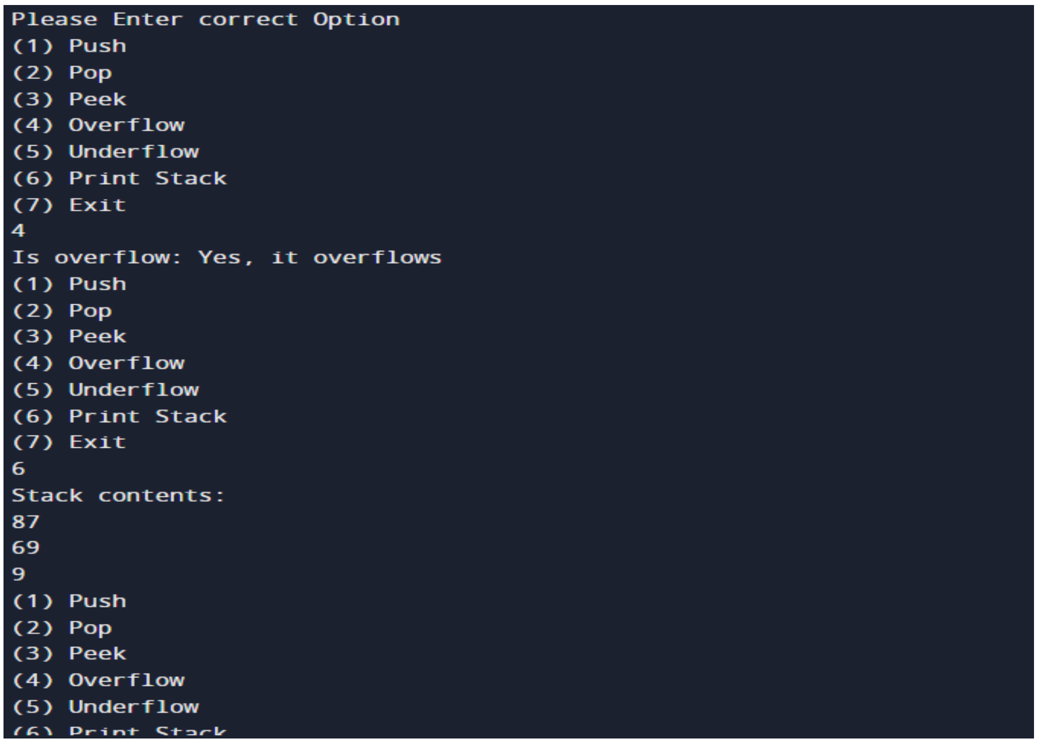
```
Enter size of your stack: 3
(1) Push
(2) Pop
(3) Peek
(4) Overflow
(5) Underflow
(6) Print Stack
(7) Exit
1
Enter an element to push: 5
(1) Push
(2) Pop
(3) Peek
(4) Overflow
(5) Underflow
(6) Print Stack
(7) Exit
2
Element 5 is popped
(1) Push
(2) Pop
(3) Peek
(4) Overflow
(5) Underflow
(6) Print Stack
(7) Exit
2
Stack empty, can't pop any more elements
```



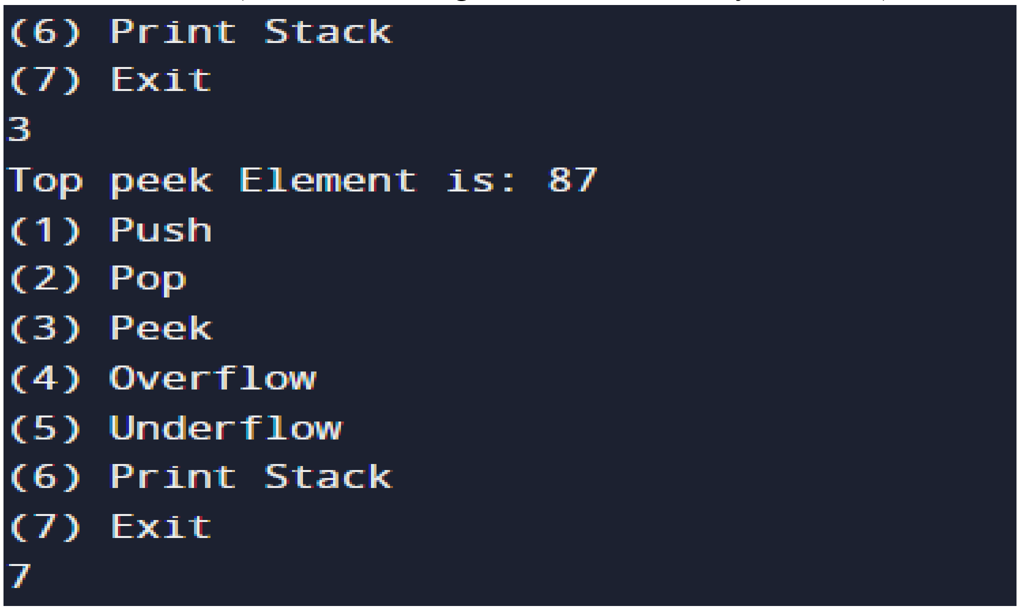
```
Element 0 is popped
(1) Push
(2) Pop
(3) Peek
(4) Overflow
(5) Underflow
(6) Print Stack
(7) Exit
5
Is it empty: Yes it's empty
(1) Push
(2) Pop
(3) Peek
(4) Overflow
(5) Underflow
(6) Print Stack
(7) Exit
1
Enter an element to push: 9
(1) Push
(2) Pop
(3) Peek
(4) Overflow
(5) Underflow
(6) Print Stack
(7) Exit
1
```



```
Enter an element to push: 69
(1) Push
(2) Pop
(3) Peek
(4) Overflow
(5) Underflow
(6) Print Stack
(7) Exit
1
Enter an element to push: 87
(1) Push
(2) Pop
(3) Peek
(4) Overflow
(5) Underflow
(6) Print Stack
(7) Exit
1
Enter an element to push: 84
Stack Full, can't push more elements(1) Push
(2) Pop
(3) Peek
(4) Overflow
(5) Underflow
(6) Print Stack
(7) Exit
47
```



```
Please Enter correct Option
(1) Push
(2) Pop
(3) Peek
(4) Overflow
(5) Underflow
(6) Print Stack
(7) Exit
4
Is overflow: Yes, it overflows
(1) Push
(2) Pop
(3) Peek
(4) Overflow
(5) Underflow
(6) Print Stack
(7) Exit
6
Stack contents:
87
69
9
(1) Push
(2) Pop
(3) Peek
(4) Overflow
(5) Underflow
(6) Print Stack
```



```
(6) Print Stack
(7) Exit
3
Top peek Element is: 87
(1) Push
(2) Pop
(3) Peek
(4) Overflow
(5) Underflow
(6) Print Stack
(7) Exit
7
```

Applications of Stack: Some applications of stack in computer science and programming include:

Function Call Stack: Stacks are used for managing function calls in programming languages. The call stack is used for keeping track of the active functions and their local variables, allowing for proper execution of the function and return.

Expression Evaluation: Stacks can be used for evaluating expressions, most importantly for those involving parentheses, operators, and operands.Example:Infix to Postfix/Prefix.

Backtracking Algorithms: Stacks are a necessity in backtracking algorithms for tasks like solving puzzles, finding paths, and searching.

Undo/Redo Functionality: Stacks are used for implementing undo and redo functionality in applications where user actions need to be tracked.

Explain the Importance of the approach followed by you

The importance of the approach followed in the code is in the clarity it provides and the modularity in implementing the stack data structure. Taking in stack-related operations in functions, the readability of the code increases, and it becomes maintainable and reusable. Also, the menu-driven interface makes it user-friendly and allows easy interaction of uses with the stack.

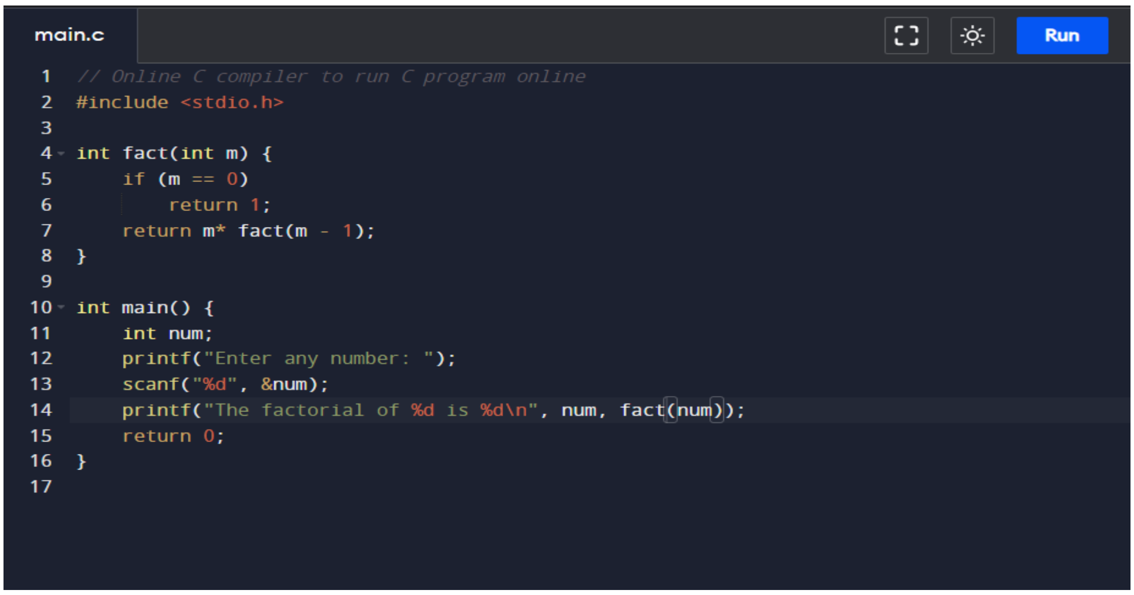
Conclusion:- To conclude, this program,performed in C programming language, offers functional implementation of a stack using an array and provides a user-friendly interface for interacting with the stack. This approach emphasizes modularity and reusability, making it a practical way for stack implementation.

PostLab Questions:

1) **Explain how Stacks can be used in Backtracking algorithms with example.**
Stacks are often used in backtracking algorithms for keeping track of choices and potential solutions. Basically,every time a decision is made, the current state is pushed onto the stack. If a decision leads to a dead end or invalid solution, the most recent state can be popped off the stack and the algorithm can backtrack to the previous state.
For example, in the "N-Queens" problem, you can use a stack to keep track of the positions where you've placed queens and backtrack when a solution becomes infeasible. Each time you backtrack, you pop a position from the stack and explore other possibilities.
2) **Illustrate the concept of Call stack in Recursion.**

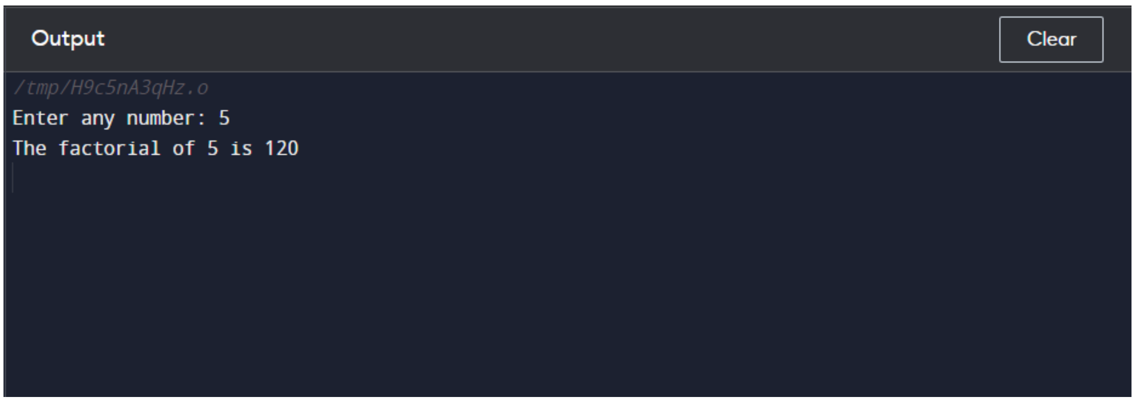
When a function calls itself recursively, the call stack is used to keep track of each active instance of the function along with its local variables and execution context. As each recursive call is made, a new entry is pushed onto the stack. When the base case is reached, the recursive calls start to return, and the stack entries are popped off, allowing the program to resume execution where it left off in each instance of the function.

For example, in a recursive function to calculate the factorial of a number:



```c
// Online C compiler to run C program online
#include <stdio.h>

int fact(int m) {
    if (m == 0)
        return 1;
    return m* fact(m - 1);
}

int main() {
    int num;
    printf("Enter any number: ");
    scanf("%d", &num);
    printf("The factorial of %d is %d\n", num, fact(num));
    return 0;
}
```

The output looks like:



```
/tmp/H9c5nA3qHz.o
Enter any number: 5
The factorial of 5 is 120
```

Logic:Since we called fact(5), the call stack would look like this:

fact (5)

fact (4)

fact (3)

fact (2)

fact (1)

fact (0)

As the base case is hit, the returns start to happen:

fact(5) returns 5 * fact(4)*fact(3)*fact(2)

fact(4) returns 4 * fact(3)*fact(2)

fact(3) returns 3 * fact(2)

fact(2) returns 2*fact(1)

fact(1) returns 1*fact(0)

fact(0) returns 1

This process continues, and the final result is calculated step by step with the stack unwinding.