|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Programming in C** | **Semester:** | **II** |
| **Date of Performance:** | **17 / 04 / 2025** | **DIV/ Batch No:** | **C4-1** |
| **Student Name:** | **Dhruv Pankhania** | **Roll No:** | **16010124216** |

**Experiment No: 9**

**Title: Implementation of Stack Data Structure Using Arrays**

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| Write a program in C to implement the stack data structure using arrays and perform basic stack operations such as push, pop, peek, and display. |

|  |
| --- |
| **COs to be achieved:** |
| **CO:** Apply basic concepts of C programming for problem-solving.(CO1 and CO2), file handling (CO5) |

|  |
| --- |
| **Theory:** |
| A stack is a linear data structure that follows the **Last In, First Out (LIFO)** principle. It allows insertion (push) and deletion (pop) operations at one end, called the **top** of the stack.   1. **Basic Stack Operations:**    * **Push(x):** Adds an element x to the top of the stack.    * **Pop():** Removes and returns the top element from the stack.    * **Peek():** Returns the top element without removing it.    * **isEmpty():** Checks if the stack is empty.    * **isFull():** Checks if the stack is full (in case of an array implementation).    * **Display():** Shows all elements in the stack. 2. **Applications of Stack:**    * Expression evaluation (infix to postfix conversion, postfix evaluation)    * Function call management (recursion)    * Undo/Redo operations in applications    * Backtracking algorithms (Maze solving, Depth First Search)   Procedure:   1. **Initialize the Stack:**    * Define an array of fixed size.    * Initialize top = -1 (indicating an empty stack). 2. **Implement Push Operation:**    * Check if the stack is full (top == size-1).    * If not, increment top and insert the element. 3. **Implement Pop Operation:**    * Check if the stack is empty (top == -1).    * If not, remove and return the top element, then decrement top. 4. **Implement Peek Operation:**    * Return the element at the top without removing it. 5. **Implement Display Operation:**    * Print all elements from top to 0. 6. **Test the Stack Implementation:**    * Perform multiple push and pop operations.    * Validate the expected outputs. |

|  |
| --- |
| **Problem Statements:** |
| Develop a program to implement a stack data structure using arrays. The program should support the following operations:   1. **Push(x):** Insert an element x into the stack. 2. **Pop():** Remove and return the top element from the stack. 3. **Peek():** Display the top element without removing it. 4. **isEmpty():** Check whether the stack is empty. 5. **isFull():** Check whether the stack is full. 6. **Display():** Print all elements present in the stack.  Constraints:  * The stack should be implemented using a fixed-size array. * The stack follows the **Last In, First Out (LIFO)** principle. * The program should handle cases of **stack overflow** (pushing into a full stack) and **stack underflow** (popping from an empty stack).  Input/Output Format:  * **Input:** The user should be able to select operations and enter values accordingly. * **Output:** Display the stack status after each operation, including error messages if applicable.   Enter stack size: 5  Choose operation: 1-Push, 2-Pop, 3-Peek, 4-Display, 5-Exit  1  Enter value to push: 10  1  Enter value to push: 20  4  Stack elements: [10, 20]  3  Top element: 20  2  Popped element: 20  4  Stack elements: [10]  5  Exiting program... |

|  |
| --- |
| **Code :** |
| #include <stdio.h>  #define MAX\_SIZE 100  *int* stack[MAX\_SIZE];  *int* top = -1;  *int* n;  *void* push(*int* *x*);  *int* pop();  *int* peek();  *void* display();  *int* main()  {    printf("Enter the stack size (Max: %d): ", MAX\_SIZE);    scanf("%d", &n);    if (n <= 0 || n > MAX\_SIZE)    {      printf("Invalid stack size.\n");      return 1;    }  *int* choice, x;    while (1)    {      printf("\n1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit\n");      printf("Enter your choice: ");      scanf("%d", &choice);      switch (choice)      {      case 1:        printf("Enter element to push: ");        scanf("%d", &*x*);        push(x);        break;      case 2:        x = pop();        if (x != -1)          printf("Popped element: %d\n", x);        break;      case 3:        x = peek();        if (x != -1)          printf("Top element: %d\n", x);        break;      case 4:        display();        break;      case 5:        printf("Exiting program.\n");        return 0;      default:        printf("Invalid choice, try again.\n");      }    }  }  *void* push(*int* *x*)  {    if (top >= n - 1)    {      printf("Stack Overflow\n");    }    else    {      stack[++top] = *x*;      printf("%d pushed to stack\n", *x*);    }  }  *int* pop()  {    if (top < 0)    {      printf("Stack Underflow\n");      return -1;    }    else    {      return stack[top--];    }  }  *int* peek()  {    if (top < 0)    {      printf("Stack is empty\n");      return -1;    }    else    {      return stack[top];    }  }  *void* display()  {    if (top < 0)    {      printf("Stack is empty\n");    }    else    {      printf("Stack contents (top to bottom):\n");      for (*int* i = top; i >= 0; i--)      {        printf("%d\n", stack[i]);      }    }  } |

|  |
| --- |
| **Output:** |
|  |

|  |
| --- |
| **Conclusion:** |
| In conclusion, we successfully implemented a simple stack using a fixed-size array in C, following the Last In, First Out (LIFO) principle. We ensured the program handles both overflow and underflow conditions effectively, making it a reliable and fundamental example of stack operations. This exercise helped us understand the core logic behind stack management, preparing us for more complex data structures ahead. |

|  |
| --- |
| **Signature of faculty in-charge with Date:** |