Q. Create a fixed memory implementation of stack. Implement push and pop operations. Display the contents of the stack after every operation.

Ans.

#include<stdio.h>

#include<stdlib.h>

struct Stack

{

    int top;

    int max\_stack;

    int\* array;

};

//function to create a stack with fixed no. of elements i.e. max\_stack

struct Stack\* createStack( int max\_stack )

{

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    if (!stack)

        return NULL;

    stack->top = -1;

    stack->max\_stack = max\_stack;

    stack->array = (int\*)malloc(stack->max\_stack \* sizeof(int));

    return stack;

}

int isEmpty(struct Stack\* stack)

{

    return stack->top == -1 ;

}

int peek(struct Stack\* stack)

{

    if (!isEmpty(stack))

        return stack->array[stack->top] ;

    return 0;

}

int pop(struct Stack\* stack)

{

    if (!isEmpty(stack))

        return stack->array[stack->top--] ;

    return 0;

}

void push(struct Stack\* stack, int op)

{

    if(stack->top+1 == stack->max\_stack){

        printf("\nStack Overflow!!\n");

    }

    else{

        stack->array[++stack->top] = op;

    }

}

void display\_stack(struct Stack\* stack){

    int len = stack->top;

    if(len == -1){

        printf("\nStack is Empty!!");

    }

    else{

        printf("\nCurrent Stack is: ");

        for(; len>-1; len--){

            printf("%d ", stack->array[len]);

        }

    }

    printf("\n");

}

int main(){

    int len, v;

    char ch;

    printf("Enter length of your required Stack: ");

    scanf("%d", &len);

    struct Stack \*stack = createStack(len);

    if(stack == NULL){

        printf("Could not create stack, memory insufficient!");

    }

    else{

        do

        {

            printf("\n1. Push to Stack: ");

            printf("\n2. Pop from Stack: ");

            printf("\n3. Peek the Stack: ");

            printf("\n4. Display the entire Stack: ");

            printf("\n5. Exit");

            printf("\nEnter your choice(1-5): ");

            fflush(stdin);

            scanf("%c", &ch);

            switch(ch)

            {

            case '1':

                printf("\nEnter the value to push: ");

                fflush(stdin);

                scanf("%d", &v);

                push(stack, v);

                display\_stack(stack);

                break;

            case '2':

                v = pop(stack);

                printf("\nValue popped is: %d", v);

                display\_stack(stack);

                break;

            case '3':

                v = peek(stack);

                printf("\nTop most element of Stack is: %d", v);

                display\_stack(stack);

                break;

            case '4':

                display\_stack(stack);

                break;

            case '5':

                printf("\nPROGRAM TERMINATED!!");

                break;

            default :

                printf("\nWrong Entry!!\n");

                break;

            }

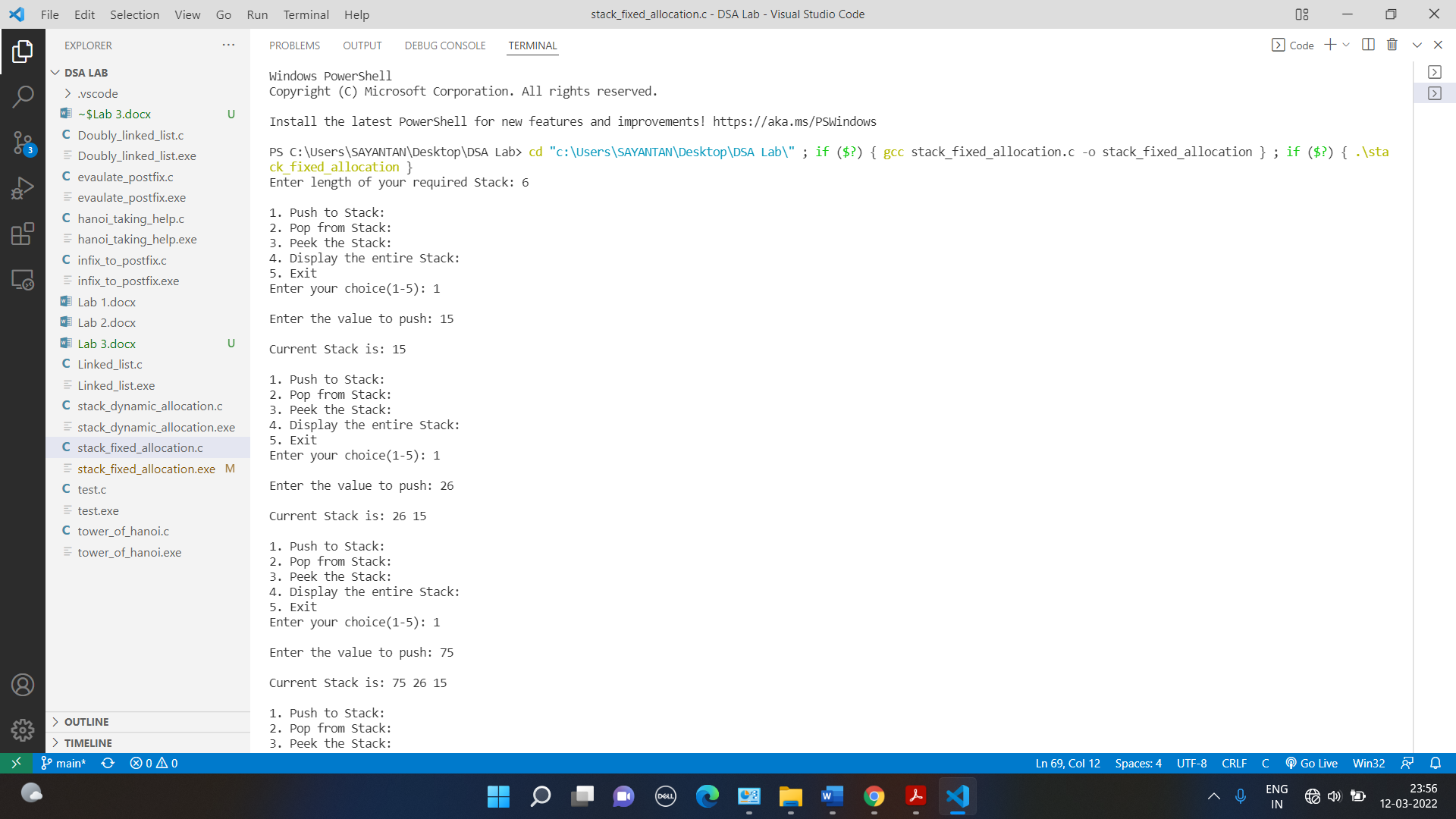
        }while(ch!='5');

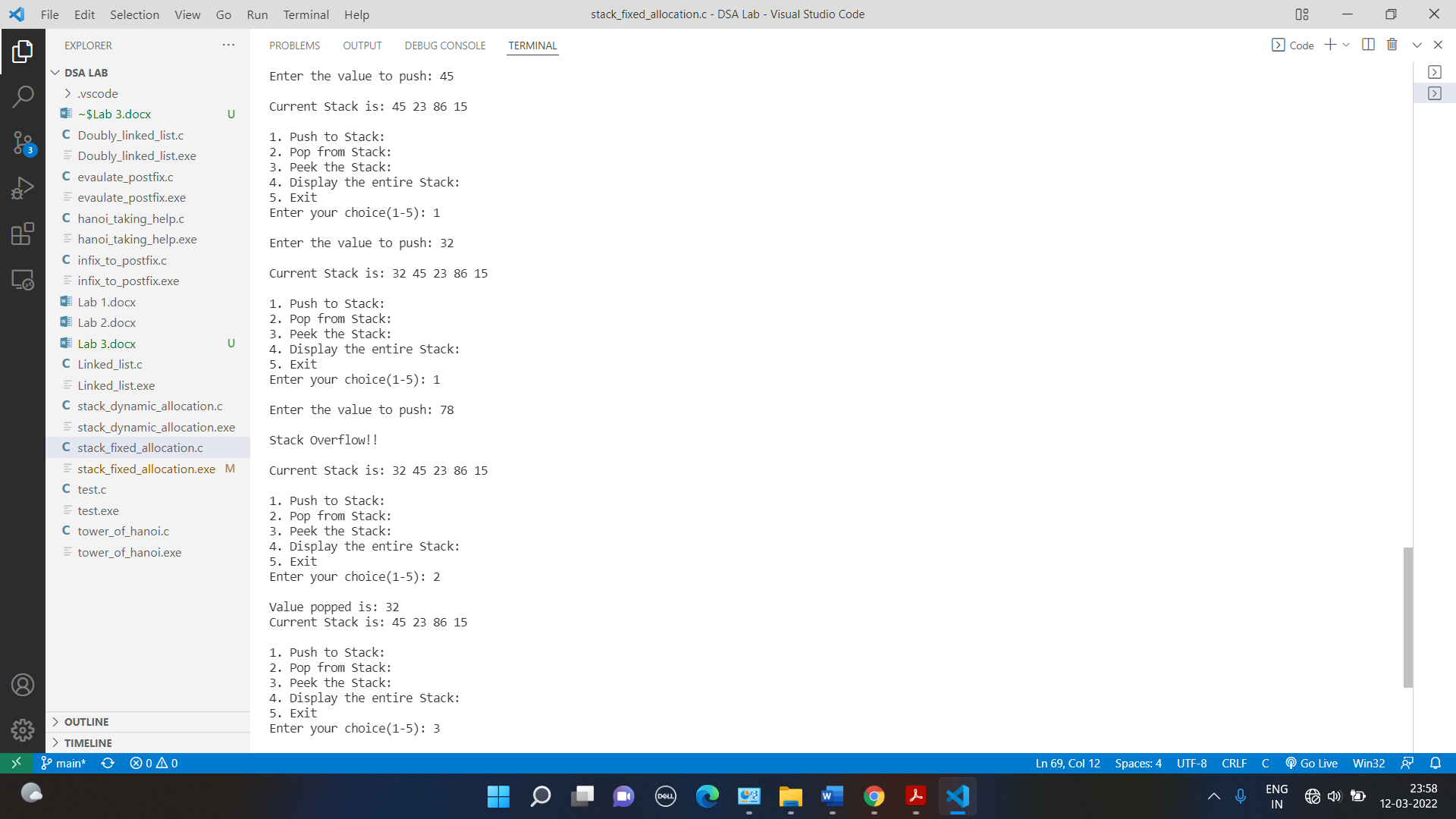
    }

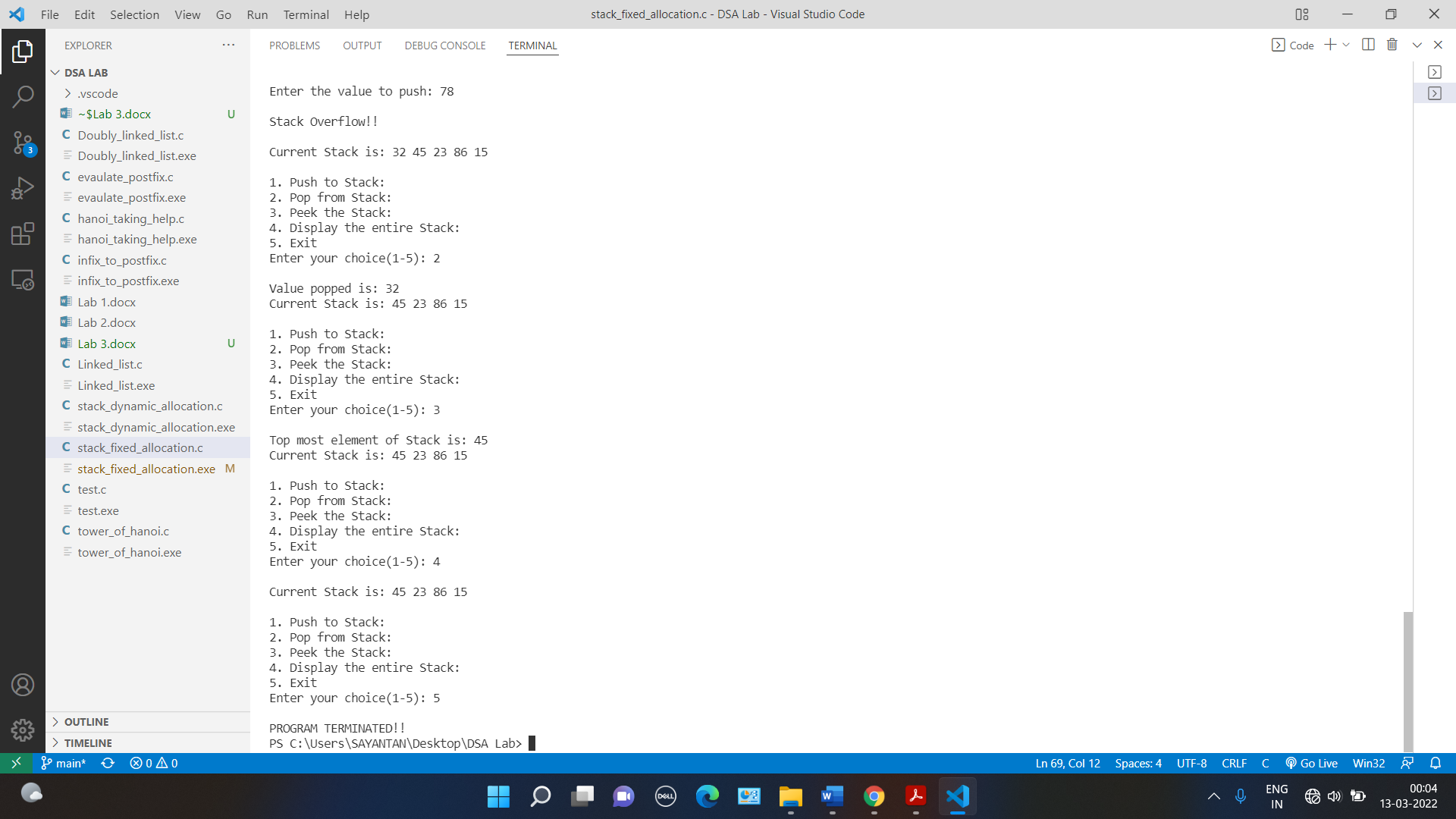
    return EXIT\_SUCCESS;

}

Output







Q. Create a dynamic memory implementation of stack. Implement push and pop operations. Display the contents of the stack after every operation.

Ans.

#include<stdio.h>

#include<stdlib.h>

struct Node{

    int val;

    struct Node\* next;

};

struct Stack

{

    int top;

    struct Node\* head;

};

struct Stack\* createStack()   //dynamic memory stack

{

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    if (!stack)

        return NULL;

    stack->top = -1;

    stack->head = NULL;

    return stack;

}

int isEmpty(struct Stack\* stack)

{

    return stack->top == -1 ;

}

int peek(struct Stack\* stack)

{

    return stack->head->val;

}

int pop(struct Stack\* stack)

{

    int store;

    if (!isEmpty(stack)){

        struct Node\* temp = stack->head;

        stack->head = stack->head->next;

        store = temp->val;

        free(temp);

        stack->top--;

        return store;

    }

    return '$';

}

void push(struct Stack\* stack, int op)

{

    struct Node\* store = (struct Node\*)malloc(sizeof(struct Node));

    if(store){

        store->next = NULL;

        store->val = op;

        if (!isEmpty(stack)){

            store->next = stack->head;

            stack->head = store;

        }

        else{

            stack->head = store;

        }

        stack->top++;

    }

    else{

        printf("Out of memory!");

    }

}

void display\_stack(struct Stack\* stack)

{

    struct Node\* store = (struct Node\*)malloc(sizeof(struct Node));

    if(store){

        store = stack->head;

        if(store == NULL){

            printf("\nStack is Empty!!");

        }

        else{

            printf("\nCurrent Stack is: ");

            while(store!= NULL){

                printf("%d ", store->val);

                store = store->next;

            }

        }

        printf("\n");

    }

    else{

        printf("\nOut of memory!");

    }

}

int main(){

    int v;

    char ch;

    struct Stack \*stack = createStack();

    if(stack == NULL){

        printf("Could not create stack!");

    }

    else{

        do

        {

            printf("\n1. Push to Stack: ");

            printf("\n2. Pop from Stack: ");

            printf("\n3. Peek the Stack: ");

            printf("\n4. Display the entire Stack: ");

            printf("\n5. Exit");

            printf("\nEnter your choice(1-5): ");

            fflush(stdin);

            scanf("%c", &ch);

            switch(ch)

            {

            case '1':

                printf("\nEnter the value to push: ");

                fflush(stdin);

                scanf("%d", &v);

                push(stack, v);

                display\_stack(stack);

                break;

            case '2':

                v = pop(stack);

                printf("\nValue popped is: %d", v);

                display\_stack(stack);

                break;

            case '3':

                v = peek(stack);

                printf("\nTop most element of Stack is: %d", v);

                display\_stack(stack);

                break;

            case '4':

                display\_stack(stack);

                break;

            case '5':

                printf("\nPROGRAM TERMINATED!!");

                break;

            default :

                printf("\nWrong Entry!!\n");

                break;

            }

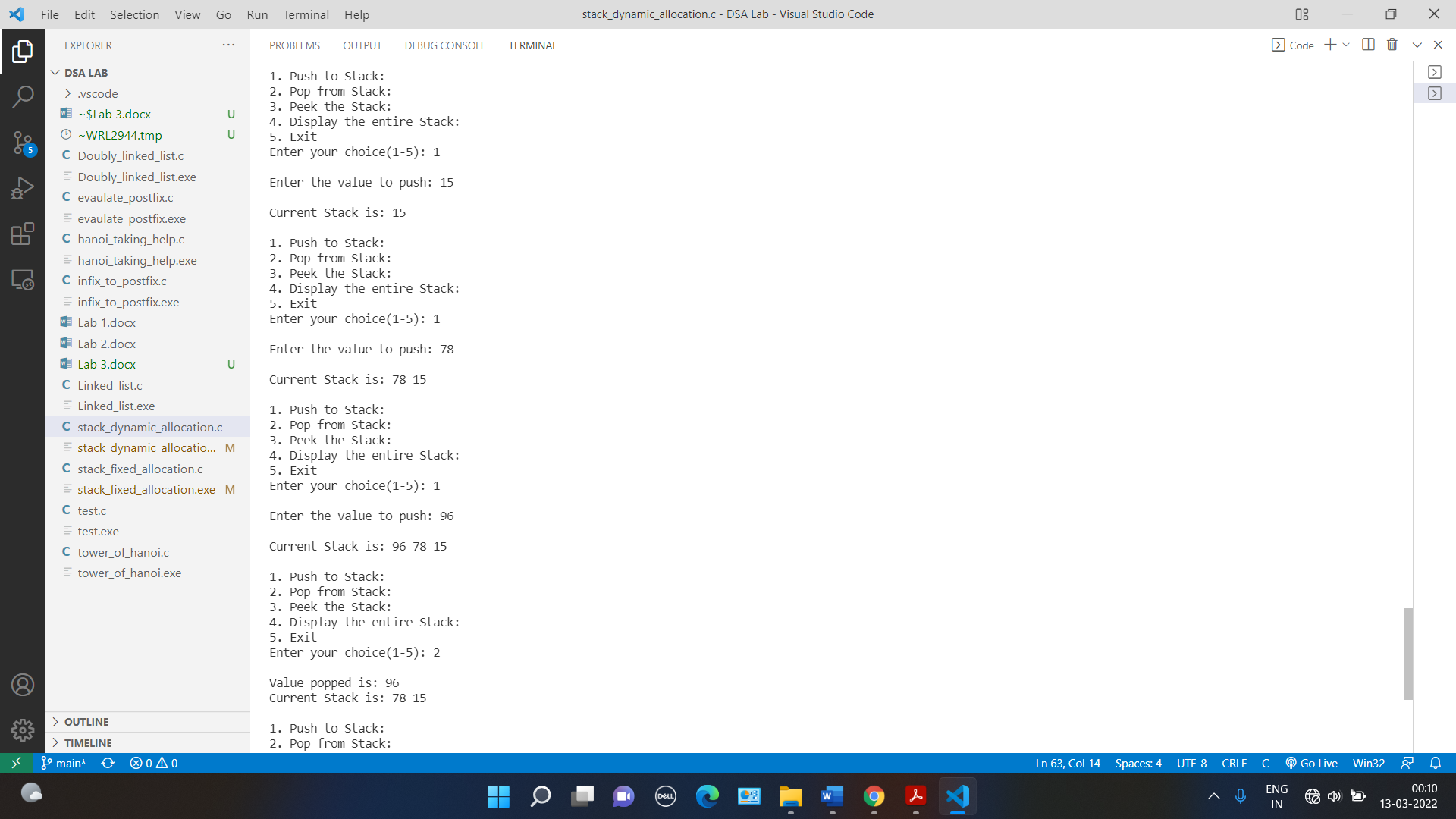
        }while(ch!='5');

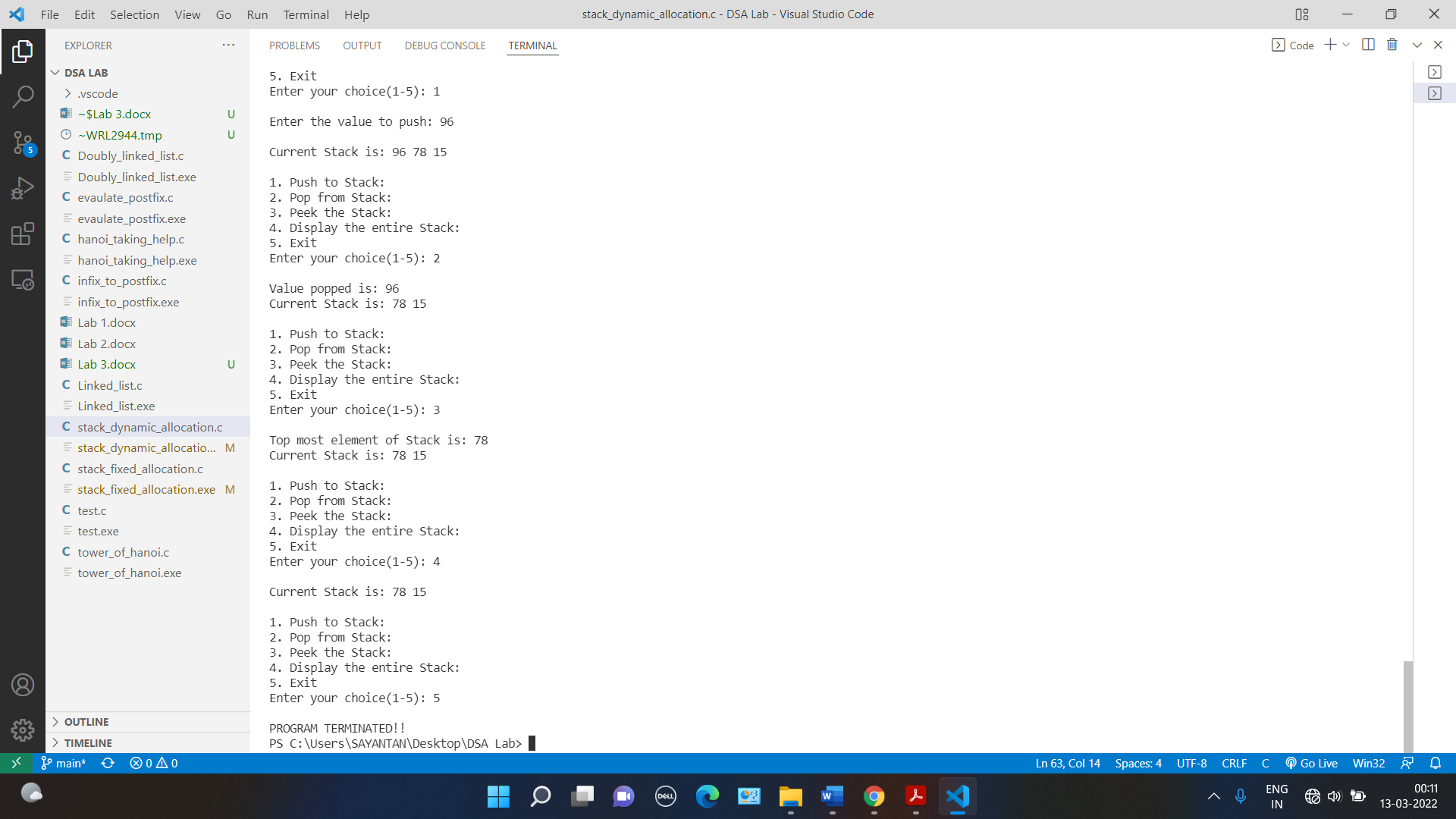
    }

    return EXIT\_SUCCESS;

}

Output





Q. Implement the Tower of Hanoi problem. You should properly enumerate the atomic operations/steps. Vary the number of discs (to be shifted).

Ans.

#include<stdio.h>

#include<stdlib.h>

void hanoi(int disk\_no, char from, char aux, char to){

    if(disk\_no!=0)

    {

        hanoi(disk\_no-1, from, to, aux);

        printf("Disk %d is moved from %c to %c\n", disk\_no, from, to);

        hanoi(disk\_no-1, aux, from, to);

    }

}

int main(){

    int n;

    printf("Enter the number of disks: ");

    scanf(" %d", &n);

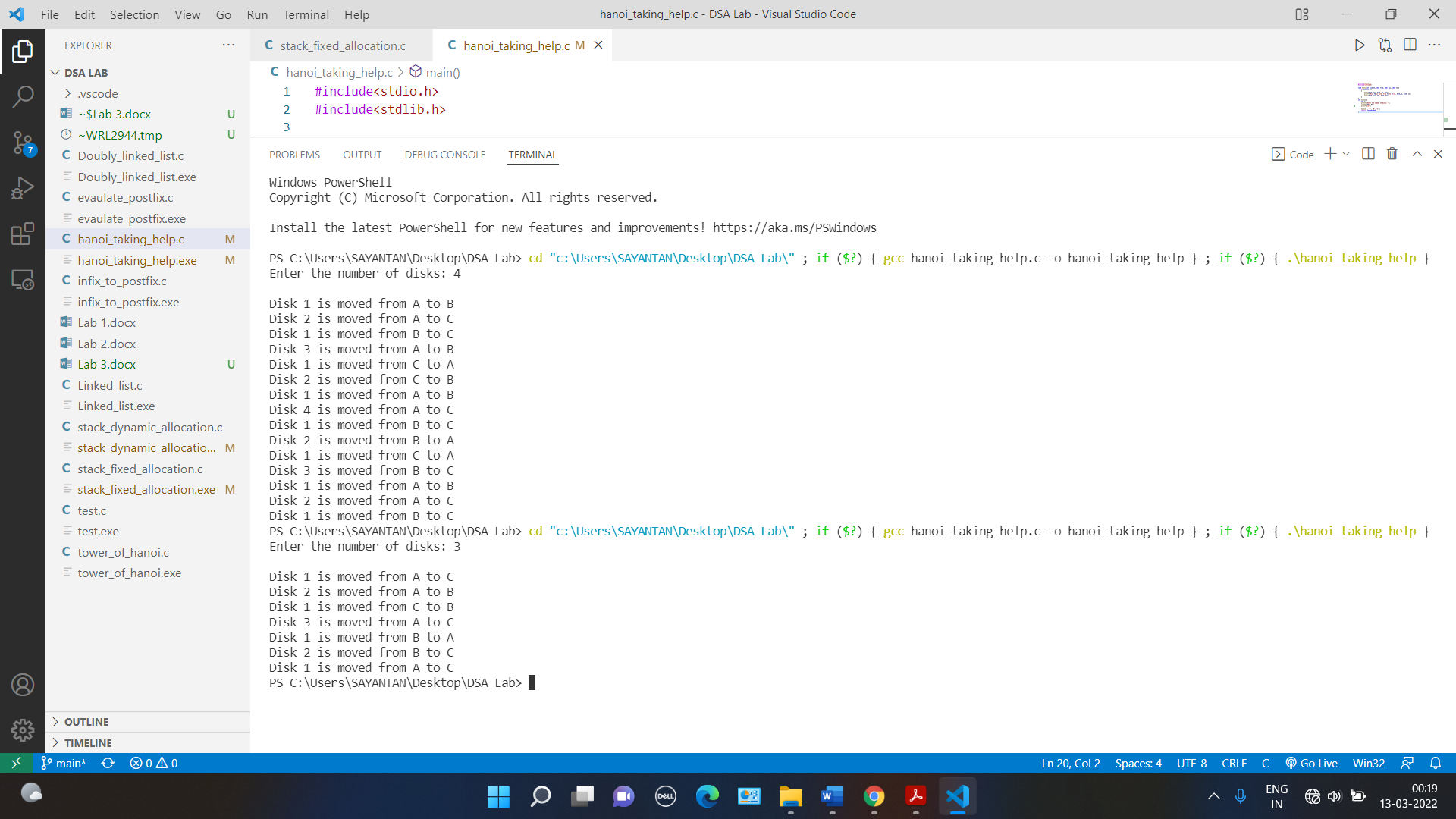
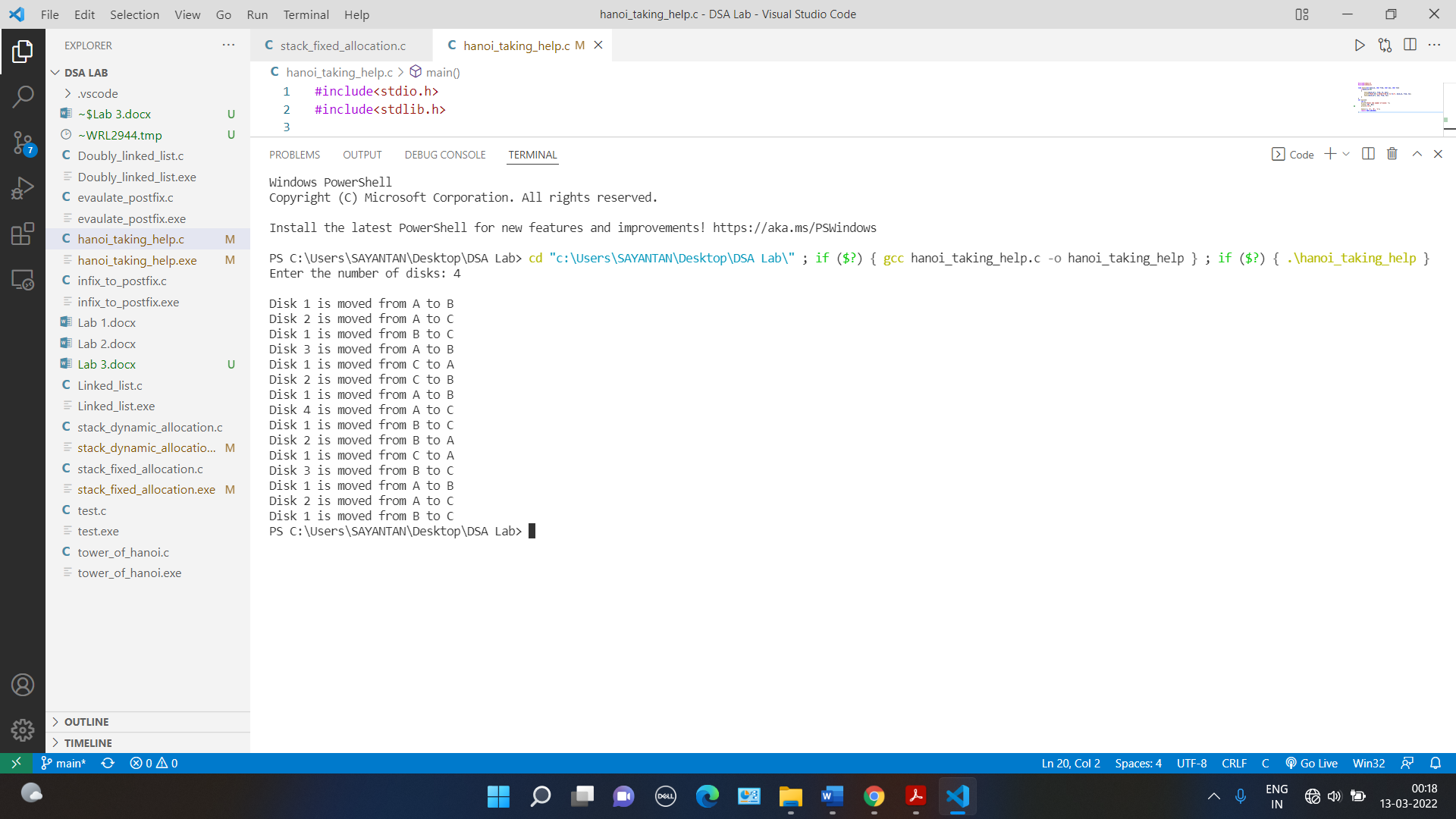
printf("\n");

    hanoi( n, 'A', 'B', 'C');

    return EXIT\_SUCCESS;

}

Output

Q. Write a program to evaluate postfix expressions using Stack. You can use $ as the end of input.

Ans.

#include<stdio.h>

#include<stdlib.h>

#include<ctype.h>

#include<math.h>

struct Node{

    int val;

    struct Node\* next;

};

struct Stack

{

    int top;

    struct Node\* head;

};

struct Stack\* createStack()   //dynamic memory stack

{

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    if (!stack)

        return NULL;

    stack->top = -1;

    stack->head = NULL;

    return stack;

}

int isEmpty(struct Stack\* stack)

{

    return stack->top == -1 ;

}

int peek(struct Stack\* stack)

{

    return stack->head->val;

}

int pop(struct Stack\* stack)

{

    int store;

    if (!isEmpty(stack)){

        struct Node\* temp = stack->head;

        stack->head = stack->head->next;

        store = temp->val;

        free(temp);

        stack->top--;

        return store;

    }

    return '$';

}

void push(struct Stack\* stack, int op)

{

    struct Node\* store = (struct Node\*)malloc(sizeof(struct Node));

    if(store){

        store->next = NULL;

        store->val = op;

        if (!isEmpty(stack)){

            store->next = stack->head;

            stack->head = store;

        }

        else{

            stack->head = store;

        }

        stack->top++;

    }

    else{

        printf("Out of memory!");

    }

}

int main(){

    int op1, op2, flag = 1;

    char ch;

    printf("Enter your postfix expression (end it with $): ");

    struct Stack \*stack = createStack();

    if(stack == NULL){

        printf("Could not create stack!");

    }

    else{

        do{

            scanf("%c", &ch);

            if(isalnum(ch)){

                push(stack, ch - '0');

            }

            else if(ch == '$')

                break;

            else{  //this means ch is an operation symbol

                op1 = pop(stack);

                op2 = pop(stack);

                if(op2 == '$'){

                    flag = 0;

                    break;

                }

                switch (ch)

                {

                case '+':

                    op2 = op2 + op1;

                    break;

                case '-':

                    op2 = op2 - op1;

                    break;

                case '/':

                    op2 = op2/op1;

                    break;

                case '\*':

                    op2 = op2 \* op1;

                    break;

                case '^':

                    op2 = pow(op2, op1);

                    break;

                default:

                    break;

                }

                push(stack, op2);

            }

        }while(ch!='$');

        if(flag == 1){

            printf("Answer: %d", pop(stack));

        }

        else{

            printf("Invalid expression!");

        }

    }

    return EXIT\_SUCCESS;

}

Output

