```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Input expression to check
char arr[] = {'(', '(', 'a', '+', 'b', ')', '*', '(', 'c', '-', 'd',
')',')',')','\0'};

// Structure Declaration
struct Stack {
    int size;
    int top;
    char *S;
} st;

// Function Prototypes
void create(struct Stack *st);
void push(struct Stack *st, char exp);
char pop(struct Stack *st);
int isEmpty(struct Stack *st);

int main() {
    // Create the stack
    create(&st);

    for (int i = 0; arr[i] != '\0'; i++) {
        if (arr[i] == '(') {
            push(&st, arr[i]);
        } else if (arr[i] == ')') {
            if (isEmpty(&st)) {
                printf("\nThe Expression is not balanced...\n");
                free(st.S);
                return 0;
            }
            pop(&st);
        }
    }

    // Final check for unbalanced parentheses
    if (isEmpty(&st)) {
        printf("\nThe Expression is balanced...\n");
    } else {
        printf("\nThe Expression is not balanced...\n");
    }

    // Free the allocated memory
    free(st.S);
    return 0;
```

```
}

// Function to create the Stack
void create(struct Stack *st) {
    st->size = strlen(arr);
    st->top = -1;
    st->S = (char *)malloc(st->size * sizeof(char)); // Allocate memory for
characters
}

// Function to push an element onto the stack
void push(struct Stack *st, char exp) {
    if (st->top == st->size - 1) {
        printf("\nStack Overflow!!!\n");
    } else {
        st->S[++st->top] = exp;
    }
}

// Function to pop an element from the stack
char pop(struct Stack *st) {
    if (isEmpty(st)) {
        printf("\nStack Underflow or Empty!!\n");
        return '\0';
    } else {
        return st->S[st->top--];
    }
}

// Function to check if the stack is empty
int isEmpty(struct Stack *st) {
    return st->top == -1;
}
```

2.

```
//Infix to postfix

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//Structure for stack declaration :
struct Stack
{
    int size;
    int top;
    char *S;
}st;
```

```c
//Function Prototype
void create(struct Stack *,char []);
void infixToPostfix(struct Stack *,char []);
void push(struct Stack *, char);
char pop(struct Stack *);
int precedence(char);
int isEmpty(struct Stack *);


int main(){
    char inarr[20];
    printf("\nEnter the infix expression : ");
    scanf("%[^\n]s",inarr);


    //Creating the stack for the purpose of conversion :
    create(&st,inarr);


    //Function call for converting the infix to postfix :
    infixToPostfix(&st,inarr);



    return 0;
}
void create(struct Stack *st,char inarr[]) {

    //Setting the size of the stack :
    st->size = strlen(inarr);

    //Initializing the top of the stack to -1 : as empty stack.
    st->top = -1;


    st->S = (char *)malloc(st->size * sizeof(char)); // Allocate memory for
characters
}

void infixToPostfix(struct Stack *st,char inarr[])
{
    char postarr[50];
    int k = 0;
    for(int i=0;inarr[i]!='\0';i++)
    {
        if(inarr[i]=='(')
        {
```

```
            push(st,inarr[i]);
        }
        //if the character is an operand then it should be added to the
postarr[].
        else if((inarr[i]>='a' && inarr[i]<='z')||(inarr[i]>='A' &&
inarr[i]<='Z'))
        {
            postarr[k++] = inarr[i];
        }
        else if(inarr[i]==')')
        {
            //pop until '(' is found
            while (!isEmpty(st) && st->S[st->top] != '(' )
            {
                postarr[k++] = pop(st);
            }
            pop(st); //pop '('
        }
        //if the character is an operator :
        else{
            while (!isEmpty(st) && precedence(st->S[st->top]) >=
precedence(inarr[i]))
            {
                postarr[k++] = pop(st);
            }
            push(st,inarr[i]);
        }
    }
    while(!isEmpty(st)){
        postarr[k++] = pop(st);
    }
    postarr[k] = '\0';//For NULL TERMINATING the postfix character array

    printf("\nThe postfix expression obtained after conversion is :
%s\n",postarr);
}
// Function to push an element onto the stack
void push(struct Stack *st, char exp) {
    if (st->top == st->size - 1) {
        printf("\nStack Overflow!!!\n");
    } else {
        st->S[++st->top] = exp;
    }
}

// Function to pop an element from the stack
char pop(struct Stack *st) {
    if (isEmpty(st)) {
```

```
        printf("\nStack Underflow or Empty!!\n");
        return '\0';
    } else {
        return st->S[st->top--];
    }
}
// Function to check if the stack is empty
int isEmpty(struct Stack *st) {
    return st->top == -1;
}

// Function to check precedence of operators:
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}
```

3.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structure for stack declaration
struct Stack {
    int size;
    int top;
    char *S;
}st;

// Function Prototypes
void create(struct Stack *, char[]);
void infixToPostfix(struct Stack *, char[]);
void push(struct Stack *, char);
char pop(struct Stack *);
int precedence(char);
int isEmpty(struct Stack *);
void postToPrefix(char[]);

int main() {
    char inarr[50];
    printf("\nEnter the infix expression: ");
    scanf("%[^\n]s", inarr);
    char temp[50];
    int k = 0;

    // Reverse the input string
    for (int i = strlen(inarr) - 1; i >= 0; i--) {
        // Swap '(' with ')' and vice versa
```

```
        if (inarr[i] == '(')
            temp[k++] = ')';
        else if (inarr[i] == ')')
            temp[k++] = '(';
        else
            temp[k++] = inarr[i];
    }
    temp[k] = '\0';

    printf("\nThe Reversed string is: %s", temp);

    // Create the stack for conversion
    create(&st, temp);

    // Convert the reversed infix to postfix
    infixToPostfix(&st, temp);

    return 0;
}

// Function to create the stack
void create(struct Stack *st, char inarr[]) {
    st->size = strlen(inarr);
    st->top = -1;
    st->S = (char *)malloc(st->size * sizeof(char));
}

// Function to convert infix to postfix
void infixToPostfix(struct Stack *st, char inarr[]) {
    char postarr[50];
    int k = 0;
    for (int i = 0; inarr[i] != '\0'; i++) {
        if (inarr[i] == '(') {
            push(st, inarr[i]);
        }
        else if ((inarr[i] >= 'a' && inarr[i] <= 'z') || (inarr[i] >= 'A' &&
inarr[i] <= 'Z')) {
            postarr[k++] = inarr[i];
        }
        else if (inarr[i] == ')') {
            while (!isEmpty(st) && st->S[st->top] != '(') {
                postarr[k++] = pop(st);
            }
            pop(st); // Remove '('
        }
        else {
            while (!isEmpty(st) && precedence(st->S[st->top]) >=
precedence(inarr[i])) {
```

```
                postarr[k++] = pop(st);
            }
            push(st, inarr[i]);
        }
    }
    while (!isEmpty(st)) {
        postarr[k++] = pop(st);
    }
    postarr[k] = '\0'; // Null-terminate the postfix array

    printf("\nThe postfix expression is: %s", postarr);

    // Convert postfix to prefix
    postToPrefix(postarr);
}

// Function to push an element onto the stack
void push(struct Stack *st, char exp) {
    if (st->top == st->size - 1) {
        printf("\nStack Overflow!!!\n");
    } else {
        st->S[++st->top] = exp;
    }
}

// Function to pop an element from the stack
char pop(struct Stack *st) {
    if (isEmpty(st)) {
        printf("\nStack Underflow or Empty!!\n");
        return '\0';
    } else {
        return st->S[st->top--];
    }
}

// Function to check if the stack is empty
int isEmpty(struct Stack *st) {
    return st->top == -1;
}

// Function to check precedence of operators
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}

// Function to convert postfix to prefix
```

```
void postToPrefix(char inarr[]) {
    char temp[50];
    int k = 0;

    // Reverse the postfix expression
    for (int i = strlen(inarr) - 1; i >= 0; i--) {
        temp[k++] = inarr[i];
    }
    temp[k] = '\0';

    printf("\nThe prefix expression is: %s\n", temp);
}
```

4.

```
//Reverse a string using stack

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structure for stack declaration
struct Stack {
    int size;
    int top;
    char *S;
}st;

// Function Prototypes
void create(struct Stack *, char[]);
void push(struct Stack *, char);
void reverse(struct Stack *);
int isEmpty(struct Stack *);
char pop(struct Stack *);

int main()
{
    char arr[50];
    printf("\nEnter the string to be reversed : ");
    scanf("%[^\n]s",arr);

    create(&st,arr);

    for(int i =0;arr[i]!='\0';i++)
    {
        push(&st,arr[i]);
    }
```

```
    reverse(&st);
    return 0;


}

// Function to create the stack
void create(struct Stack *st, char inarr[]) {
    st->size = strlen(inarr);
    st->top = -1;
    st->S = (char *)malloc(st->size * sizeof(char));
}

//For pushing character in the stack
void push(struct Stack *st, char exp) {
    if (st->top == st->size - 1) {
        printf("\nStack Overflow!!!\n");
    } else {
        st->S[++st->top] = exp;
    }
}


// Function to pop an element from the stack
char pop(struct Stack *st) {
    if (isEmpty(st)) {
        printf("\nStack Underflow or Empty!!\n");
        return '\0';
    } else {
        return st->S[st->top--];
    }
}


//Function for Reverse String
void reverse(struct Stack *st){
    char temp[50];
    int k=0;
    while (!isEmpty(st)) {
        temp[k++] = pop(st);
    }
    temp[k] = '\0'; // Null-terminate the postfix array

    printf("\nThe Reversed string  is: %s", temp);

}
int isEmpty(struct Stack *st) {
    return st->top == -1;
}
```

5.

```c
//Queue Concept : Enqueue and Dequeue operations are Done in this program.
#include <stdio.h>
#include <stdlib.h>

#define MAX 100 // Maximum size of the queue

// Structure for the Queue
typedef struct {
    int arr[MAX];
    int front; // Index of the front element
    int rear;  // Index of the rear element
} Queue;

// Function prototypes
void initializeQueue(Queue *q);
int isFull(Queue *q);
int isEmpty(Queue *q);
void enqueue(Queue *q, int value);
int dequeue(Queue *q);
void displayQueue(Queue *q);

int main() {
    Queue q;
    initializeQueue(&q);

    int choice, value;
    while (1) {
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(&q, value);
                break;

            case 2:
                value = dequeue(&q);
                if (value != -1) {
                    printf("Dequeued value: %d\n", value);
                }
```

```
                break;

            case 3:
                displayQueue(&q);
                break;

            case 4:
                printf("Exiting...\n");
                exit(0);

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}

// Function to initialize the queue
void initializeQueue(Queue *q) {
    q->front = -1;
    q->rear = -1;
}

// Function to check if the queue is full
int isFull(Queue *q) {
    return q->rear == MAX - 1;
}

// Function to check if the queue is empty
int isEmpty(Queue *q) {
    return q->front == -1 || q->front > q->rear;
}

// Function to enqueue (add) an element to the queue
void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full! Cannot enqueue.\n");
        return;
    }
    if (q->front == -1) {
        q->front = 0; // Initialize front when the first element is added
    }
    q->arr[++q->rear] = value;
    printf("Enqueued: %d\n", value);
}

// Function to dequeue (remove) an element from the queue
```

```c
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty! Cannot dequeue.\n");
        return -1;
    }
    int value = q->arr[q->front++];
    // Reset the queue if all elements are dequeued
    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }
    return value;
}

// Function to display the queue
void displayQueue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = q->front; i <= q->rear; i++) {
        printf("%d ", q->arr[i]);
    }
    printf("\n");
}
```

6.

```c
/*
1.Simulate a Call Center Queue
Create a program to simulate a call center where incoming calls are handled on
a first-come, first-served basis.
Use a queue to manage call handling and provide options to add, remove, and
view calls.

*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 100 // Maximum size of the queue

// Structure for the Call Center Queue
typedef struct {
    char calls[MAX][50]; // Array to store call details (e.g., phone numbers
or caller names)
    int front;           // Index of the first call in the queue
```

```c
    int rear;            // Index of the last call in the queue
} CallQueue;

// Function prototypes
void initializeQueue(CallQueue *q);
int isFull(CallQueue *q);
int isEmpty(CallQueue *q);
void addCall(CallQueue *q, char *caller);
void handleCall(CallQueue *q);
void viewQueue(CallQueue *q);

int main() {
    CallQueue callQueue;
    initializeQueue(&callQueue);

    int choice;
    char caller[50];

    while (1) {
        printf("\nCall Center Queue Operations:\n");
        printf("1. Add Call\n");
        printf("2. Handle Call\n");
        printf("3. View Queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar(); // Clear the input buffer

        switch (choice) {
            case 1:
                printf("Enter caller's name or phone number: ");
                fgets(caller, 50, stdin);
                caller[strcspn(caller, "\n")] = '\0'; // Remove the newline
character
                addCall(&callQueue, caller);
                break;

            case 2:
                handleCall(&callQueue);
                break;

            case 3:
                viewQueue(&callQueue);
                break;

            case 4:
                printf("Exiting the Call Center system. Goodbye!\n");
                exit(0);
```

```
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}

// Function to initialize the queue
void initializeQueue(CallQueue *q) {
    q->front = -1;
    q->rear = -1;
}

// Function to check if the queue is full
int isFull(CallQueue *q) {
    return q->rear == MAX - 1;
}

// Function to check if the queue is empty
int isEmpty(CallQueue *q) {
    return q->front == -1 || q->front > q->rear;
}

// Function to add a call to the queue
void addCall(CallQueue *q, char *caller) {
    if (isFull(q)) {
        printf("Queue is full! Cannot add more calls.\n");
        return;
    }
    if (q->front == -1) {
        q->front = 0; // Initialize front when the first call is added
    }
    strcpy(q->calls[++q->rear], caller);
    printf("Added call from: %s\n", caller);
}

// Function to handle (remove) a call from the queue
void handleCall(CallQueue *q) {
    if (isEmpty(q)) {
        printf("No calls to handle! The queue is empty.\n");
        return;
    }
    printf("Handling call from: %s\n", q->calls[q->front++]);
    // Reset the queue if all calls are handled
    if (q->front > q->rear) {
        q->front = q->rear = -1;
```

```
    }
}

// Function to view the current queue of calls
void viewQueue(CallQueue *q) {
    if (isEmpty(q)) {
        printf("No calls in the queue!\n");
        return;
    }
    printf("Current call queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("%d. %s\n", i - q->front + 1, q->calls[i]);
    }
}
```

7.

```
/*
2.Print Job Scheduler
Implement a print job scheduler where print requests are queued. Allow users
to add new print jobs, cancel a specific job, and print jobs in the order they
were added.


*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 100 // Maximum size of the queue

// Structure for a Print Job
typedef struct {
    int jobID;
    char jobName[50];
} PrintJob;

// Structure for the Print Job Queue
typedef struct {
    PrintJob jobs[MAX]; // Array to store print jobs
    int front;
    int rear;
} PrintQueue;

// Function prototypes
void initializeQueue(PrintQueue *q);
int isFull(PrintQueue *q);
```

```c
int isEmpty(PrintQueue *q);
void addPrintJob(PrintQueue *q, int jobID, char *jobName);
void cancelJob(PrintQueue *q, int jobID);
void processPrintJob(PrintQueue *q);
void viewPrintQueue(PrintQueue *q);

int main() {
    PrintQueue printQueue;
    initializeQueue(&printQueue);

    int choice, jobID;
    char jobName[50];

    while (1) {
        printf("\nPrint Job Scheduler:\n");
        printf("1. Add Print Job\n");
        printf("2. Cancel Print Job\n");
        printf("3. Process Print Job\n");
        printf("4. View Print Queue\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar(); // Clear the input buffer

        switch (choice) {
            case 1:
                printf("Enter Job ID: ");
                scanf("%d", &jobID);
                getchar(); // Clear the input buffer
                printf("Enter Job Name: ");
                fgets(jobName, 50, stdin);
                jobName[strcspn(jobName, "\n")] = '\0'; // Remove the newline
character
                addPrintJob(&printQueue, jobID, jobName);
                break;

            case 2:
                printf("Enter Job ID to cancel: ");
                scanf("%d", &jobID);
                cancelJob(&printQueue, jobID);
                break;

            case 3:
                processPrintJob(&printQueue);
                break;

            case 4:
                viewPrintQueue(&printQueue);
```

```
                break;

            case 5:
                printf("Exiting the Print Job Scheduler. Goodbye!\n");
                exit(0);

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}

// Function to initialize the queue
void initializeQueue(PrintQueue *q) {
    q->front = -1;
    q->rear = -1;
}

// Function to check if the queue is full
int isFull(PrintQueue *q) {
    return q->rear == MAX - 1;
}

// Function to check if the queue is empty
int isEmpty(PrintQueue *q) {
    return q->front == -1 || q->front > q->rear;
}

// Function to add a new print job to the queue
void addPrintJob(PrintQueue *q, int jobID, char *jobName) {
    if (isFull(q)) {
        printf("Queue is full! Cannot add more print jobs.\n");
        return;
    }
    if (q->front == -1) {
        q->front = 0; // Initialize front when the first job is added
    }
    q->jobs[++q->rear].jobID = jobID;
    strcpy(q->jobs[q->rear].jobName, jobName);
    printf("Added print job: ID = %d, Name = %s\n", jobID, jobName);
}

// Function to cancel a specific print job
void cancelJob(PrintQueue *q, int jobID) {
    if (isEmpty(q)) {
        printf("No jobs to cancel! The queue is empty.\n");
```

```
        return;
    }

    int found = 0;
    for (int i = q->front; i <= q->rear; i++) {
        if (q->jobs[i].jobID == jobID) {
            found = 1;
            for (int j = i; j < q->rear; j++) {
                q->jobs[j] = q->jobs[j + 1];
            }
            q->rear--;
            if (q->rear < q->front) {
                q->front = -1; // Reset queue if empty
                q->rear = -1;
            }
            printf("Canceled print job with ID = %d\n", jobID);
            break;
        }
    }

    if (!found) {
        printf("No print job found with ID = %d\n", jobID);
    }
}

// Function to process (remove) the first print job in the queue
void processPrintJob(PrintQueue *q) {
    if (isEmpty(q)) {
        printf("No jobs to process! The queue is empty.\n");
        return;
    }
    printf("Processing print job: ID = %d, Name = %s\n", q->jobs[q-
>front].jobID, q->jobs[q->front].jobName);
    q->front++;
    if (q->front > q->rear) {
        q->front = q->rear = -1; // Reset queue if empty
    }
}

// Function to view the current print queue
void viewPrintQueue(PrintQueue *q) {
    if (isEmpty(q)) {
        printf("No print jobs in the queue!\n");
        return;
    }
    printf("Current print jobs in the queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("ID = %d, Name = %s\n", q->jobs[i].jobID, q->jobs[i].jobName);
```

```
    }
}
```

8.

```
/*
3.Design a Ticketing System
Simulate a ticketing system where people join a queue to buy tickets.
Implement functionality for people to join the queue, buy tickets, and display
the queue's current state.
has context menu

*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 100 // Maximum queue size

// Structure for a person in the queue
typedef struct {
    int ticketID;
    char name[50];
} Person;

// Structure for the Ticket Queue
typedef struct {
    Person queue[MAX];
    int front;
    int rear;
} TicketQueue;

// Function prototypes
void initializeQueue(TicketQueue *q);
int isFull(TicketQueue *q);
int isEmpty(TicketQueue *q);
void joinQueue(TicketQueue *q, int ticketID, char *name);
void buyTicket(TicketQueue *q);
void displayQueue(TicketQueue *q);

int main() {
    TicketQueue ticketQueue;
    initializeQueue(&ticketQueue);

    int choice, ticketID;
    char name[50];
```

```c
    while (1) {
        printf("\nTicketing System:\n");
        printf("1. Join Queue\n");
        printf("2. Buy Ticket\n");
        printf("3. Display Queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar(); // Clear the input buffer

        switch (choice) {
            case 1:
                printf("Enter Ticket ID: ");
                scanf("%d", &ticketID);
                getchar(); // Clear the input buffer
                printf("Enter Name: ");
                fgets(name, 50, stdin);
                name[strcspn(name, "\n")] = '\0'; // Remove the newline
character
                joinQueue(&ticketQueue, ticketID, name);
                break;

            case 2:
                buyTicket(&ticketQueue);
                break;

            case 3:
                displayQueue(&ticketQueue);
                break;

            case 4:
                printf("Exiting the Ticketing System. Goodbye!\n");
                exit(0);

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}

// Function to initialize the queue
void initializeQueue(TicketQueue *q) {
    q->front = -1;
    q->rear = -1;
}
```

```c
// Function to check if the queue is full
int isFull(TicketQueue *q) {
    return q->rear == MAX - 1;
}

// Function to check if the queue is empty
int isEmpty(TicketQueue *q) {
    return q->front == -1 || q->front > q->rear;
}

// Function for a person to join the queue
void joinQueue(TicketQueue *q, int ticketID, char *name) {
    if (isFull(q)) {
        printf("Queue is full! Cannot join.\n");
        return;
    }
    if (q->front == -1) {
        q->front = 0; // Initialize front for the first person
    }
    q->queue[++q->rear].ticketID = ticketID;
    strcpy(q->queue[q->rear].name, name);
    printf("%s with Ticket ID %d has joined the queue.\n", name, ticketID);
}

// Function to buy a ticket (remove the first person from the queue)
void buyTicket(TicketQueue *q) {
    if (isEmpty(q)) {
        printf("No one in the queue to buy a ticket.\n");
        return;
    }
    printf("%s with Ticket ID %d has bought a ticket.\n", q->queue[q-
>front].name, q->queue[q->front].ticketID);
    q->front++;
    if (q->front > q->rear) {
        q->front = q->rear = -1; // Reset queue if empty
    }
}

// Function to display the current state of the queue
void displayQueue(TicketQueue *q) {
    if (isEmpty(q)) {
        printf("The queue is currently empty.\n");
        return;
    }
    printf("Current queue state:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("Ticket ID: %d, Name: %s\n", q->queue[i].ticketID, q-
>queue[i].name);
```

```
    }
}
```