1)C Program to Check Balanced Parentheses in Expressions Using Stack

ANSWER:

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

// Define the stack structure
struct Stack {
    int size;
    int top;
    char *S;  // Array to store stack elements
};

// Function declarations
void create(struct Stack *, int);
void push(struct Stack *, char);
char pop(struct Stack *);
int isEmpty(struct Stack *);
int isBalance(struct Stack *, char *);

int main() {
    struct Stack st;  // Declare a stack
    char exp[100];    // Array to store the expression

    // Get input from the user
    printf("Enter the expression to check balance: ");
    scanf("%s", exp);

    // Initialize the stack based on the expression length
    create(&st, strlen(exp));

    // Check if the expression is balanced and output result
    if (isBalance(&st, exp)) {
        printf("Expression is balanced\n");
    } else {
        printf("Expression is not balanced\n");
    }

    // Free allocated memory
    free(st.S);

    return 0;
}
```

```
// Function to create the stack
void create(struct Stack *st, int size) {
    st->size = size;
    st->top = -1;  // Initialize top to -1 (empty stack)
    st->S = (char *)malloc(st->size * sizeof(char));  // Allocate memory for stack
}

// Function to push a character onto the stack
void push(struct Stack *st, char x) {
    if (st->top == st->size - 1) {
        printf("Stack overflow\n");  // Stack is full
    } else {
        st->top++;
        st->S[st->top] = x;  // Add character to stack
    }
}

// Function to pop a character from the stack
char pop(struct Stack *st) {
    if (st->top == -1) {
        printf("Stack underflow\n");  // Stack is empty
        return '\0';  // Return null character
    } else {
        char x = st->S[st->top];
        st->top--;  // Decrease top to pop the element
        return x;  // Return the popped character
    }
}

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

// Function to check if the parentheses in the expression are balanced
int isBalance(struct Stack *st, char *exp) {
    int i;
    for (i = 0; exp[i] != '\0'; i++) {
        if (exp[i] == '(') {
            push(st, '(');  // Push '(' onto the stack
        } else if (exp[i] == ')') {
            if (isEmpty(st)) {
                return 0;  // Unmatched ')' found, not balanced
```

```
            }
            pop(st);  // Pop '(' from the stack
        }
    }
    return isEmpty(st) ? 1 : 0;  // Return 1 if balanced, 0 if not
}
```

2)Infix to Postfix Conversion and String Reversal Program

ANSWER:

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

// Define a structure for the stack
struct Stack {
    int size;
    int top;
    char *S;
};

// Function prototypes
void create(struct Stack *, int);
void push(struct Stack *, char);
char pop(struct Stack *);
int isEmpty(struct Stack *);
int precedence(char);
void infixtoPostfix(struct Stack*, char*, char*);
void reverseString(char*);

int main() {
    struct Stack st; // Declare a stack
    char str[20], infix[100], postfix[100], prefix[100];

    // Input the infix expression
    printf("Enter the infix expression: ");
    scanf("%s", infix);

    // Initialize stack
    create(&st, strlen(infix));

    // Convert infix expression to postfix
    infixtoPostfix(&st, infix, postfix);
```

```
    printf("Postfix expression: %s\n", postfix);

    // Input a string to reverse
    printf("Enter a string: ");
    scanf("%s", str);

    // Reverse the string
    reverseString(str);
    printf("Reversed string: %s\n", str);

    // Free allocated memory for the stack
    free(st.S);
    return 0;
}

// Function to initialize stack with a given size
void create(struct Stack *st, int size) {
    st->size = size;
    st->top = -1;
    st->S = (char *)malloc(st->size * sizeof(char));
}

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

// Function to pop an element from the stack
char pop(struct Stack *st) {
    if (st->top == -1) {
        printf("Stack underflow\n");
        return '\0';
    } else {
        char x = st->S[st->top];
        st->top--;
        return x;
    }
}
```

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

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

// Function to convert an infix expression to postfix
void infixtoPostfix(struct Stack *st, char* infix, char* postfix) {
    int i, j = 0;
    for (i = 0; infix[i] != '\0'; i++) {
        char ch = infix[i];

        // If the character is an operand (letter or number), add it to the postfix expression
        if (isalnum(ch)) {
            postfix[j++] = ch;
        }
        // If the character is '(', push it to the stack
        else if (ch == '(') {
            push(st, ch);
        }
        // If the character is ')', pop from the stack until '(' is found
        else if (ch == ')') {
            while (!isEmpty(st) && st->S[st->top] != '(') {
                postfix[j++] = pop(st);
            }
            pop(st);  // Pop the '(' from the stack
        }
        // If the character is an operator, pop operators with higher or equal precedence
from the stack
        else {
            while (!isEmpty(st) && precedence(st->S[st->top]) >= precedence(ch)) {
                postfix[j++] = pop(st);
            }
            push(st, ch);  // Push the current operator onto the stack
```

```
        }
    }

    // Pop all remaining operators from the stack
    while (!isEmpty(st)) {
        postfix[j++] = pop(st);
    }

    postfix[j] = '\0'; // Null-terminate the postfix expression
}

// Function to reverse a string in place
void reverseString(char *str) {
    int n = strlen(str);
    for (int i = 0; i < n / 2; i++) {
        // Swap characters
        char temp = str[i];
        str[i] = str[n - i - 1];
        str[n - i - 1] = temp;
    }
}
```

3)Queue Implementation Using Arrays in C

ANSWER:

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

// Define the Queue structure
struct Queue{
    int size;   // Maximum size of the queue
    int front;  // Front index of the queue
    int rear;   // Rear index of the queue
    int *Q;     // Pointer to the array that will hold queue elements
};

// Function prototypes
void enqueue(struct Queue *, int);
int dequeue(struct Queue *);

int main()
{
    struct Queue q;  // Declare a queue variable
```

```
    // Take the size of the queue from the user
    printf("Enter the size: ");
    scanf("%d", &q.size);

    // Allocate memory for the queue elements
    q.Q = (int *)malloc(q.size * sizeof(int));

    // Initialize front and rear to -1, indicating an empty queue
    q.front = q.rear = -1;

    // Enqueue some elements into the queue
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    enqueue(&q, 40);
    enqueue(&q, 50);

    // Dequeue an element and print it
    printf("Dequeued: %d\n", dequeue(&q));

    // Free the allocated memory for the queue
    free(q.Q);

    return 0;
}

// Function to add an element to the queue
void enqueue(struct Queue *q, int x)
{
    // Check if the queue is full
    if (q->rear == q->size - 1) {
        printf("Queue is full\n");
    } else {
        // Increment the rear and add the element to the queue
        q->rear++;
        q->Q[q->rear] = x;
        printf("Enqueued: %d\n", x);
    }
}

// Function to remove and return an element from the queue
int dequeue(struct Queue *p)
{
    int x = -1;
```

```
    // Check if the queue is empty
    if (p->front == p->rear) {
        printf("Queue is empty\n");
    } else {
        // Increment the front and get the element at that position
        p->front++;
        x = p->Q[p->front];
    }
    return x;  // Return the dequeued element
}
```

4).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.

ANSWER:

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

// Define the structure for storing call information
struct Call {
    int id;               // Unique call ID
    char callerName[50];   // Name of the caller
};

// Define the Queue structure for managing the calls
struct Queue {
    int size;         // Maximum size of the queue
    int front;        // Front index of the queue
    int rear;         // Rear index of the queue
    struct Call *Q;    // Pointer to the array that holds the calls
};

// Function prototypes
void enqueue(struct Queue *, int, char*);
struct Call dequeue(struct Queue *);
void display(struct Queue *);

int main() {
    struct Queue q;  // Declare a queue variable
```

```
    printf("Enter the size of queue: ");
    scanf("%d", &q.size);  // Take the size of the queue as input

    // Allocate memory for the queue to hold 'size' number of calls
    q.Q = (struct Call*)malloc(q.size * sizeof(struct Call));

    // Initialize front and rear to -1, indicating an empty queue
    q.front = q.rear = -1;

    int choice, call = 0;  // Initialize choice and call counter
    char callerName[50];   // Array to store the caller's name

    while(1) {
        // Display menu options
        printf("1. Add call (Enqueue)\n");
        printf("2. Remove call (Dequeue)\n");
        printf("3. View calls\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:  // Enqueue a new call
                if (q.rear == q.size - 1) {
                    printf("Queue is full\n");  // Check if queue is full
                } else {
                    printf("Enter caller name: ");
                    scanf("%s", callerName);  // Get caller's name
                    enqueue(&q, call++, callerName);  // Enqueue the call
                }
                break;

            case 2:  // Dequeue a call
                if (q.front == q.rear) {
                    printf("Queue is empty. No calls to handle.\n");  // If queue is empty
                } else {
                    struct Call handledCall = dequeue(&q);  // Dequeue the call
                    printf("Handled call ID: %d, Caller name: %s\n", handledCall.id,
handledCall.callerName);
                }
                break;

            case 3:  // Display all calls in the queue
                display(&q);  // Call the display function
```

```
            break;

        case 4:  // Exit the program
            free(q.Q);  // Free the allocated memory for the queue
            printf("Exiting the queue\n");
            return 0;

        default:
            printf("Invalid choice\n");  // Handle invalid input
      }
   }

   return 0;
}

// Function to enqueue a call (add a call to the queue)
void enqueue(struct Queue *q, int id, char *callerName) {
   q->rear++;  // Increment the rear index
   q->Q[q->rear].id = id;  // Set the call ID
   strcpy(q->Q[q->rear].callerName, callerName);  // Copy the caller name into the
queue
   printf("Added call ID %d, Caller Name: %s\n", id, callerName);  // Confirm the
addition
}

// Function to dequeue a call (remove a call from the queue)
struct Call dequeue(struct Queue *q) {
   q->front++;  // Increment the front index
   return q->Q[q->front];  // Return the call at the front of the queue
}

// Function to display all calls in the queue
void display(struct Queue *q) {
   if (q->front == q->rear) {
      printf("Queue is empty.\n");  // If the queue is empty
   } else {
      printf("Calls in queue:\n");
      for (int i = q->front + 1; i <= q->rear; i++) {
         printf("Caller ID: %d, Caller name: %s\n", q->Q[i].id, q->Q[i].callerName);  //
Display each call
      }
   }
}
```

5).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.

ANSWER:

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

struct PrintJob {
    int id;
    char jobName[50];
};

struct Queue {
    int size;
    int front;
    int rear;
    struct PrintJob *Q;
};

void addJob(struct Queue *, int, char *);
void cancelJob(struct Queue *, int);
void viewJobs(struct Queue *);
int isEmpty(struct Queue *);

int main() {
    struct Queue q;
    printf("Enter the number of print jobs: ");
    scanf("%d", &q.size);
    q.Q = (struct PrintJob *)malloc(q.size * sizeof(struct PrintJob));
    q.front = q.rear = -1;

    int choice, jobId = 0;
    char jobName[50];

    while (1) {
        printf("\n1. Add Print Job\n");
        printf("2. Cancel Print Job\n");
        printf("3. View Pending Jobs\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1:
                if (q.rear == q.size - 1) {
                    printf("Queue is full. Cannot add more jobs.\n");
                } else {
                    printf("Enter job name: ");
                    scanf(" %49s", jobName); // Limit input to avoid buffer overflow
                    addJob(&q, ++jobId, jobName);
                }
                break;

            case 2:
                if (isEmpty(&q)) {
                    printf("Queue is empty. No jobs to cancel.\n");
                } else {
                    int cancelId;
                    printf("Enter the job ID to cancel: ");
                    scanf("%d", &cancelId);
                    cancelJob(&q, cancelId);
                }
                break;

            case 3:
                viewJobs(&q);
                break;

            case 4:
                free(q.Q);
                printf("Exiting the Print Job Scheduler.\n");
                return 0;

            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
    return 0;
}

int isEmpty(struct Queue *q) {
    return q->front == q->rear;
}

void addJob(struct Queue *q, int id, char *jobName) {
    q->rear++;
```

```
    q->Q[q->rear].id = id;
    strncpy(q->Q[q->rear].jobName, jobName, sizeof(q->Q[q->rear].jobName) - 1);
    q->Q[q->rear].jobName[sizeof(q->Q[q->rear].jobName) - 1] = '\0'; // Ensure null
termination
    printf("Added Print Job ID: %d, Job Name: %s\n", id, jobName);
}

void cancelJob(struct Queue *q, int id) {
    int found = 0;
    for (int i = q->front + 1; i <= q->rear; i++) {
        if (q->Q[i].id == id) {
            found = 1;
            printf("Cancelled Print Job ID: %d, Job Name: %s\n", q->Q[i].id,
q->Q[i].jobName);

            // Shift remaining jobs
            for (int j = i; j < q->rear; j++) {
                q->Q[j] = q->Q[j + 1];
            }
            q->rear--;
            break;
        }
    }
    if (!found) {
        printf("Print Job with ID %d not found.\n", id);
    }
}

void viewJobs(struct Queue *q) {
    if (isEmpty(q)) {
        printf("No pending print jobs.\n");
    } else {
        printf("Pending Print Jobs:\n");
        for (int i = q->front + 1; i <= q->rear; i++) {
            printf("Job ID: %d, Job Name: %s\n", q->Q[i].id, q->Q[i].jobName);
        }
    }
}
```

6).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

ANSWER:

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

struct Customer {
    int id;
    char name[50];
};

struct Queue {
    int size;
    int front;
    int rear;
    struct Customer *Q;
};

void joinQueue(struct Queue *, int, char *);
struct Customer buyTicket(struct Queue *);
void displayQueue(struct Queue *);
int isQueueEmpty(struct Queue *);
int isQueueFull(struct Queue *);

int main() {
    struct Queue q;
    printf("Enter the maximum number of customers in the queue: ");
    scanf("%d", &q.size);
    q.Q = (struct Customer *)malloc(q.size * sizeof(struct Customer));
    q.front = q.rear = -1;

    int choice, customerId = 0;
    char customerName[50];

    while (1) {
        printf("\n1. Join the Queue\n");
        printf("2. Buy Ticket\n");
        printf("3. Display Queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                if (isQueueFull(&q)) {
```

```
                printf("Queue is full. No more customers can join.\n");
            } else {
                printf("Enter customer name: ");
                getchar(); // To handle newline character left by previous scanf
                scanf("%49s", customerName);
                joinQueue(&q, ++customerId, customerName);
            }
            break;
        case 2:
            if (isQueueEmpty(&q)) {
                printf("Queue is empty. No customers to buy tickets.\n");
            } else {
                struct Customer servedCustomer = buyTicket(&q);
                printf("Ticket issued to Customer ID: %d, Name: %s\n",
servedCustomer.id, servedCustomer.name);
            }
            break;
        case 3:
            displayQueue(&q);
            break;
        case 4:
            free(q.Q);
            printf("Exiting the Ticketing System.\n");
            return 0;
        default:
            printf("Invalid choice. Please try again.\n");
      }
   }
   return 0;
}

int isQueueEmpty(struct Queue *q) {
   return q->front == q->rear;
}

int isQueueFull(struct Queue *q) {
   return q->rear == q->size - 1;
}

void joinQueue(struct Queue *q, int id, char *name) {
   q->rear++;
   q->Q[q->rear].id = id;
   strncpy(q->Q[q->rear].name, name, sizeof(q->Q[q->rear].name) - 1);
   q->Q[q->rear].name[sizeof(q->Q[q->rear].name) - 1] = '\0';
```

```
    printf("Customer ID %d, Name: %s joined the queue.\n", id, name);
}

struct Customer buyTicket(struct Queue *q) {
    q->front++;
    return q->Q[q->front];
}

void displayQueue(struct Queue *q) {
    if (isQueueEmpty(q)) {
        printf("No customers in the queue.\n");
    } else {
        printf("Customers in the queue:\n");
        for (int i = q->front + 1; i <= q->rear; i++) {
            printf("Customer ID: %d, Name: %s\n", q->Q[i].id, q->Q[i].name);
        }
    }
}
```

7)Implementation of queue using linked list

ANSWER:

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

struct Node {
    int data;
    struct Node *next;
};

struct Queue {
    struct Node *front;
    struct Node *rear;
};

// Function to add an element to the queue
void enqueue(struct Queue *, int);

// Function to remove an element from the queue
int dequeue(struct Queue *);

// Function to display the current elements in the queue
void display(struct Queue *);
```

```
int main() {
    struct Queue q;
    q.front = q.rear = NULL;  // Initialize an empty queue

    // Enqueue elements
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    enqueue(&q, 40);

    // Display the current queue
    display(&q);

    // Dequeue and display the dequeued element
    printf("Dequeued: %d\n", dequeue(&q));
    printf("Dequeued: %d\n", dequeue(&q));

    // Display the queue after dequeuing elements
    display(&q);

    return 0;
}

// Enqueue function: Adds a new element at the rear of the queue
void enqueue(struct Queue *q, int value) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));  // Allocate
memory for a new node
    newNode->data = value;
    newNode->next = NULL;

    if (q->rear == NULL) {  // If the queue is empty
        q->front = q->rear = newNode;  // New node is both front and rear
    } else {
        q->rear->next = newNode;  // Attach new node at the rear
        q->rear = newNode;        // Update the rear pointer
    }
}

// Dequeue function: Removes the front element from the queue and returns its value
int dequeue(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty.\n");
        return -1;  // Return -1 if queue is empty
```

```
    }

    struct Node *temp = q->front;
    int value = temp->data;  // Store the value to be returned
    q->front = q->front->next;  // Move the front pointer to the next node

    if (q->front == NULL) {
        q->rear = NULL;  // If the queue is empty, set rear to NULL
    }

    free(temp);  // Free memory allocated to the dequeued node
    return value;
}

// Display function: Prints the elements in the queue
void display(struct Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty.\n");
        return;
    }

    struct Node *temp = q->front;
    printf("Queue: ");
    while (temp != NULL) {
        printf("%d ", temp->data);  // Print the data of each node
        temp = temp->next;  // Move to the next node
    }
    printf("\n");
}
```