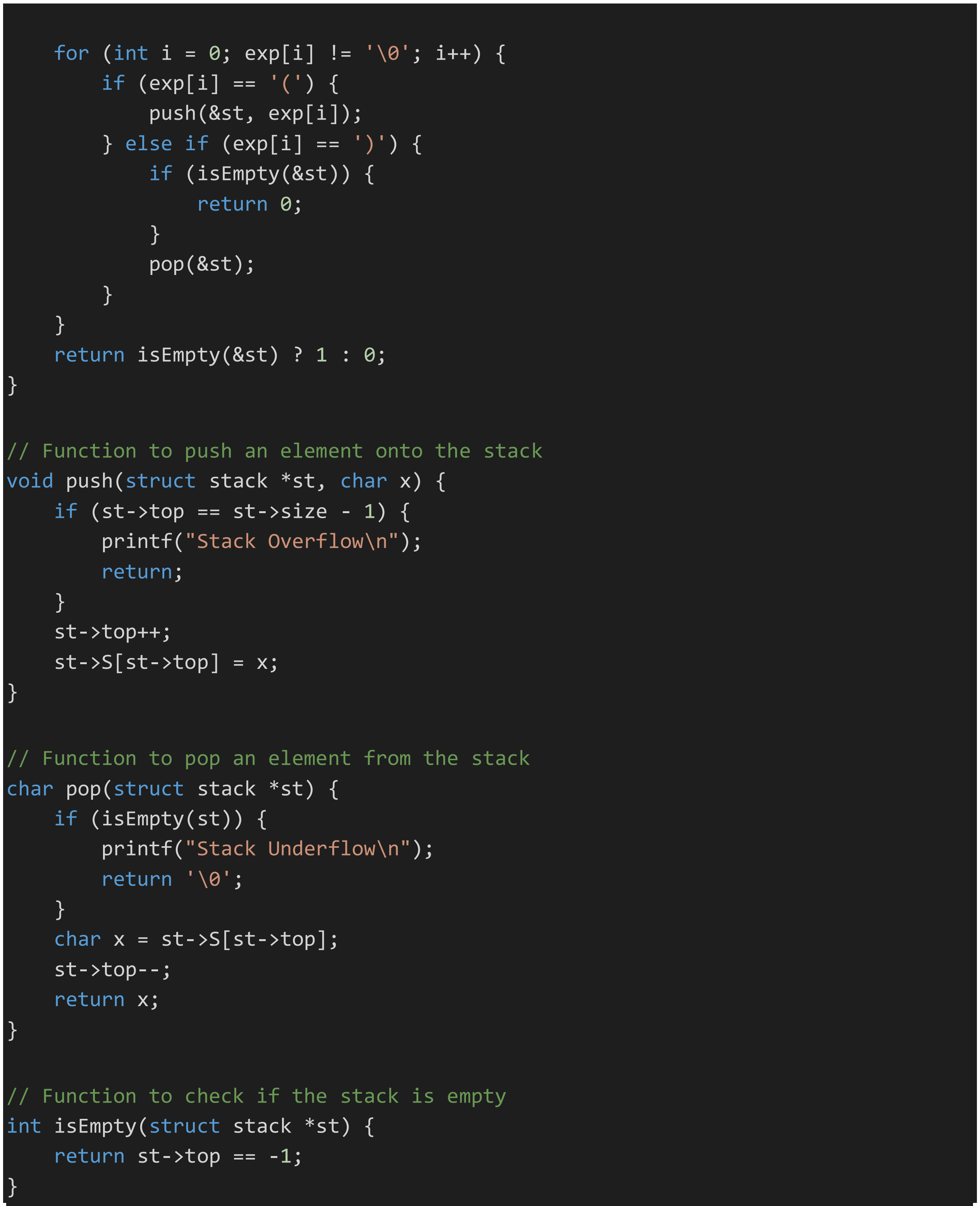
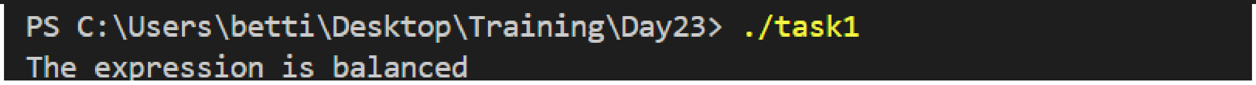
1.

```c
//implementation of checking balanced expressions

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

// Stack structure
struct stack {
    int size;
    int top;
    char *S;
};

// Function prototypes
int isBalance(char *exp);
void push(struct stack *st, char x);
char pop(struct stack *st);
int isEmpty(struct stack *st);

int main() {
    char exp[] = "((a+b)*(c-d))";

    if (isBalance(exp)) {
        printf("The expression is balanced\n");
    } else {
        printf("The expression is not balanced\n");
    }

    return 0;
}

// Function to check if the expression is balanced
int isBalance(char *exp) {
    struct stack st;
    st.size = strlen(exp);
    st.top = -1;
    st.S = (char *)malloc(st.size * sizeof(char));
    if (st.S == NULL) {
        printf("Memory allocation failed\n");
        return 0;
    }
```



```c
    for (int i = 0; exp[i] != '\0'; i++) {
        if (exp[i] == '(') {
            push(&st, exp[i]);
        } else if (exp[i] == ')') {
            if (isEmpty(&st)) {
                return 0;
            }
            pop(&st);
        }
    }
    return isEmpty(&st) ? 1 : 0;
}

// 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");
        return;
    }
    st->top++;
    st->S[st->top] = x;
}

// Function to pop an element from the stack
char pop(struct stack *st) {
    if (isEmpty(st)) {
        printf("Stack Underflow\n");
        return '\0';
    }
    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;
}
```



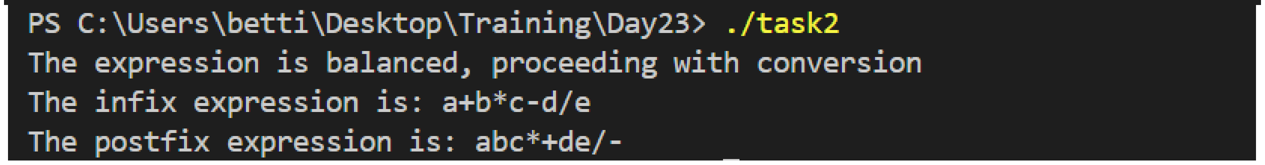
```
PS C:\Users\betti\Desktop\Training\Day23> ./task1
The expression is balanced
```

2.

```c
//implementation of infix to postfix conversion

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

// Stack structure
struct stack {
    int size;
    int top;
    char *S;
};

struct stack st;

// Function prototypes
int isBalance(char *);
char* removeParentheses(char *);
void push(struct stack *, char);
char pop(struct stack *);
int isEmpty(struct stack *);
int precedence(char op);
char* inToPost(char *);



int main() {
    char exp[] = "((a+(b*c))-(d/e))";

    if (!isBalance(exp)) {
        printf("The expression is not balanced, cannot proceed\n");
    } else {
        printf("The expression is balanced, proceeding with conversion\n");
        char *infix = removeParentheses(exp);
        char *postfix = inToPost(infix);
        printf("The infix expression is: %s\n", infix);
        printf("The postfix expression is: %s\n", postfix);
        free(infix);
        free(postfix);
        free(st.S);
    }

    return 0;
```

```c
}

// Function to check if the expression is balanced
int isBalance(char *exp) {
    st.size = strlen(exp);
    st.top = -1;
    st.S = (char *)malloc(st.size * sizeof(char));
    if (st.S == NULL) {
        printf("Memory allocation failed\n");
        return 0;
    }

    for (int i = 0; exp[i] != '\0'; i++) {
        if (exp[i] == '(') {
            push(&st, exp[i]);
        } else if (exp[i] == ')') {
            if (isEmpty(&st)) {
                return 0;
            }
            pop(&st);
        }
    }
    int balanced = isEmpty(&st);
    free(st.S);
    return balanced;
}

// 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");
        return;
    }
    st->top++;
    st->S[st->top] = x;
}

// Function to pop an element from the stack
char pop(struct stack *st) {
    if (isEmpty(st)) {
        printf("Stack Underflow\n");
        return '\0';
    }
    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 remove parentheses from the expression
char* removeParentheses(char *exp) {
    int length = strlen(exp);

    char *result = (char *)malloc(length + 1);
    if (result == NULL) {
        printf("Memory allocation failed\n");
        return NULL;
    }

    int j = 0;
    for (int i = 0; i < length; i++) {
        if (exp[i] != '(' && exp[i] != ')') {
            result[j++] = exp[i];
        }
    }
    result[j] = '\0';

    return result;
}

// Function to find precedence of an operator
int precedence(char op) {
    if (op == '+' || op == '-') {
        return 1;
    } else if (op == '*' || op == '/') {
        return 2;
    } else if (op == '^') {
        return 3;
    }
    return -1;
}
```

```c
// Function to convert infix expression to postfix expression
char* inToPost(char *exp) {
    char *p = exp;
    char *post = (char *)malloc(strlen(exp) * sizeof(char));
    if(post == NULL) {
        printf("Memory allocation failed\n");
        return NULL;
    }

    int j = 0;
    for(int i = 0; i < strlen(exp); i++) {
        if(isalnum(p[i])) {
            post[j++] = p[i];
        } else {
            while(!isEmpty(&st) && precedence(p[i]) <= precedence(st.S[st.top]))
{
                post[j++] = pop(&st);
            }
            push(&st, p[i]);
        }
    }

    while(!isEmpty(&st)) {
        post[j++] = pop(&st);
    }
    post[j] = '\0';

    return post;
}
```



```
PS C:\Users\betti\Desktop\Training\Day23> ./task2
The expression is balanced, proceeding with conversion
The infix expression is: a+b*c-d/e
The postfix expression is: abc*+de/-
```

3.

```c
//implementation of infix to postfix conversion

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

// Stack structure
```

```c
struct stack {
    int size;
    int top;
    char *S;
};

struct stack st;

// Function prototypes
int isBalance(char *);
void push(struct stack *, char);
char pop(struct stack *);
int isEmpty(struct stack *);
int precedence(char op);
char* inToPost(char *);


int main() {
    char exp[] = "((a+(b*c))-(d/e))";

    if (!isBalance(exp)) {
        printf("The expression is not balanced, cannot proceed\n");
    } else {
        printf("The expression is balanced, proceeding with conversion\n");
        char *postfix = inToPost(exp);
        printf("The infix expression is: %s\n", exp);
        printf("The postfix expression is: %s\n", postfix);
        free(postfix);
        free(st.S);
    }

    return 0;
}

// Function to check if the expression is balanced
int isBalance(char *exp) {
    st.size = strlen(exp);
    st.top = -1;
    st.S = (char *)malloc(st.size * sizeof(char));
    if (st.S == NULL) {
        printf("Memory allocation failed\n");
        return 0;
    }
    for (int i = 0; exp[i] != '\0'; i++) {
        if (exp[i] == '(') {
            push(&st, exp[i]);
        } else if (exp[i] == ')') {
            if (isEmpty(&st)) {
                return 0;
            }
            pop(&st);
        }
    }
    int balanced = isEmpty(&st);
    free(st.S);
    return balanced;
}

// 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");
        return;
    }
    st->top++;
    st->S[st->top] = x;
}

// Function to pop an element from the stack
char pop(struct stack *st) {
    if (isEmpty(st)) {
        printf("Stack Underflow\n");
        return '\0';
    }
    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 find precedence of an operator
```

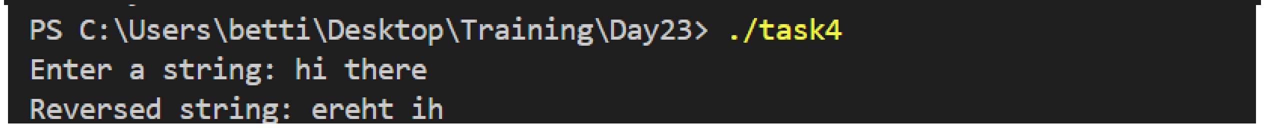
```c
int precedence(char op) {
    if (op == '+' || op == '-') {
        return 1;
    } else if (op == '*' || op == '/') {
        return 2;
    } else if (op == '^') {
        return 3;
    }
    return -1;
}

// Function to convert infix expression to postfix expression
char* inToPost(char *exp) {
    char *p = exp;
    char *post = (char *)malloc(strlen(exp) * sizeof(char));
    if(post == NULL) {
        printf("Memory allocation failed\n");
        return NULL;
    }

    int j = 0;
    for(int i = 0; i < strlen(exp); i++) {
        if(exp[i] == ')' || exp[i] == '(') {
            continue;
        }
        else if(isalnum(p[i])) {
            post[j++] = p[i];
        } else {
            while(!isEmpty(&st) && precedence(p[i]) <= precedence(st.S[st.top]))
{
                post[j++] = pop(&st);
            }
            push(&st, p[i]);
        }
    }

    while(!isEmpty(&st)) {
        post[j++] = pop(&st);
    }
    post[j] = '\0';

    return post;
}
```

```
The expression is balanced, proceeding with conversion
The infix expression is: ((a+(b*c))-(d/e))
The postfix expression is: abc*+de/-
```

4.

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

// Stack structure
struct stack {
    int size;
    int top;
    char *S;
};

// Global stack instance
struct stack st;

// Function prototypes
void create(struct stack *, char *);
void push(struct stack *, char);
char pop(struct stack *);
int isEmpty(struct stack *);
char* reverse(struct stack *);

int main() {
    char *str;
    printf("Enter a string: ");
    scanf("%[^\n]", str);

    create(&st, str);

    char *rev = reverse(&st);
    printf("Reversed string: %s\n", rev);

    free(st.S);
    return 0;
}

// Function to create a stack from an array
```

```c
void create(struct stack *st, char *str) {
    st->size = strlen(str);
    st->top = -1;
    st->S = (char *)malloc(st->size * sizeof(char));
    if (st->S == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    for (int i = 0; i < st->size; i++) {
        push(st, str[i]);
    }
}

// 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");
        return;
    }
    st->S[++(st->top)] = x;
}

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

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

char* reverse(struct stack *st) {
    char *rev = (char *)malloc((st->size + 1) * sizeof(char));
    rev[st->size] = '\0';
    for (int i = 0; i < st->size; i++) {
        rev[i] = pop(st);
    }
    return rev;
}
```



```
PS C:\Users\betti\Desktop\Training\Day23> ./task4
Enter a string: hi there
Reversed string: ereht ih
```

5.

```c
//implementation of Queue using arrays

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

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

struct Queue q;

//function prototypes
void enQueue(struct Queue *, int);
int deQueue(struct Queue *);
void createQueue(struct Queue *, int);
int isEmpty(struct Queue *);
int isFull(struct Queue *);
void displayQueue(struct Queue *);

int main() {
    int size;
    printf("Enter the size of the queue: ");
    scanf("%d", &size);
    createQueue(&q, size);

    int choice;
    while (1) {
        printf("\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                if (isFull(&q))
```

```c
                    printf("Queue is full.\n");
                else  {
                    int num;
                    printf("Enter the number to enqueue: ");
                    scanf("%d", &num);
                    enQueue(&q, num);
                }
                break;
            case 2:
                if (isEmpty(&q))
                    printf("Queue is empty.\n");
                else {
                    int dequeued = deQueue(&q);
                    printf("Dequeued element: %d\n", dequeued);
                }
                break;
            case 3:
                displayQueue(&q);
                break;
            case 4:
                free(q.Q);
                exit(0);
            default:
                printf("Invalid choice.\n");
        }
    }
    return 0;
}

void createQueue(struct Queue *q, int size) {
    q->size = size;
    q->front = -1;
    q->rear = -1;
    q->Q = (int*)malloc(size * sizeof(int));
    if (q->Q == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
}

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

```c
int isEmpty(struct Queue *q) {
    return q->front == -1 || q->front > q->rear;
}

void enQueue(struct Queue *q, int num) {
    if(isFull(q)) {
        printf("Queue is full. Dequeue some elements first.\n");
        return;
    }
    if (q->front == -1)
        q->front = 0;
    q->rear++;
    q->Q[q->rear] = num;
}

int deQueue(struct Queue *q) {
    if(isEmpty(q)) {
        printf("Queue is empty. No elements to dequeue.\n");
        return -1;
    }
    int dequeued = q->Q[q->front];
    q->front++;
    if (q->front > q->rear) {
        q->front = -1;
        q->rear = -1;
    }
    return dequeued;
}

void displayQueue(struct 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->Q[i]);
    }
    printf("\n");
}
```

```
PS C:\Users\betti\Desktop\Training\Day23> ./task5

Enter the size of the queue: 3
```

```
1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter the number to enqueue: 1


1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter the number to enqueue: 2


1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter the number to enqueue: 3


1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Queue is full.
```

```
1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 3

Queue elements: 1 2 3


1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 2

Dequeued element: 1


1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 2

Dequeued element: 2


1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 3

Queue elements: 3
```

```
1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 2

Dequeued element: 3


1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 3

Queue is empty.


1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 4
```

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>
```

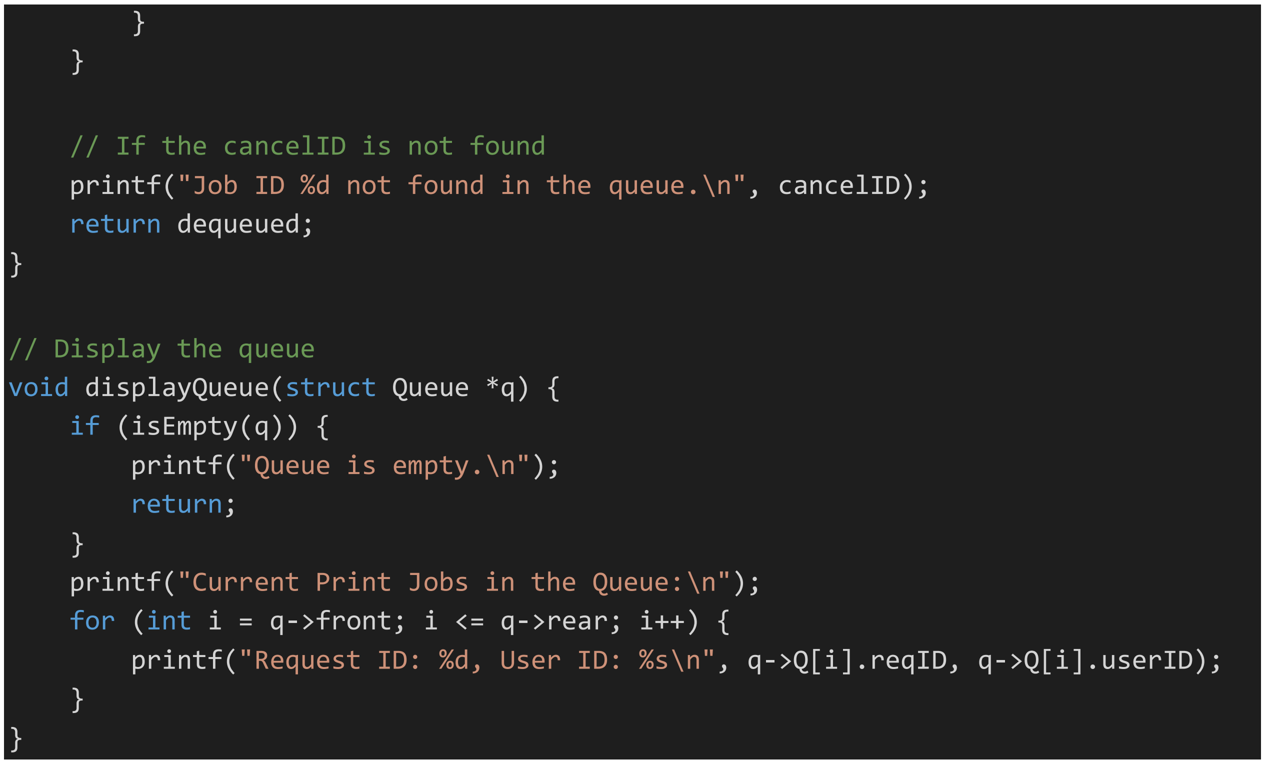
```c
// Call structure
struct Call {
    int id;
    char *caller;
};

// Queue structure
struct Queue {
    int size;
    int front;
    int rear;
    struct Call *Q;
};

// Global Queue
struct Queue q;

// Function prototypes
void createQueue(struct Queue *, int);
int isFull(struct Queue *);
int isEmpty(struct Queue *);
void enQueue(struct Queue *, int, char *);
struct Call deQueue(struct Queue *);
void displayQueue(struct Queue *);

int main() {
    int size;
    printf("Enter the total number of incoming calls: ");
    scanf("%d", &size);
    createQueue(&q, size);

    int choice;
    while (1) {
        printf("\n1. Add Call\n2. Remove Call\n3. View Calls\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                if (isFull(&q)) {
                    printf("Queue is full.\n");
                } else {
                    int id;
                    char caller[50];
                    printf("Enter the call ID: ");
                    scanf("%d", &id);
                    printf("Enter the caller's name: ");
                    scanf(" %[^\n]", caller); // Read a string with spaces
                    enQueue(&q, id, caller);
                    printf("Call added to the queue.\n");
                }
                break;
            case 2:
                if (isEmpty(&q)) {
                    printf("Queue is empty.\n");
                } else {
                    struct Call dequeued = deQueue(&q);
                    printf("Dequeued Call - ID: %d, Caller: %s\n", dequeued.id,
dequeued.caller);
                    free(dequeued.caller); // Free memory for the dequeued
caller's name
                }
                break;
            case 3:
                displayQueue(&q);
                break;
            case 4:
                // Free allocated memory
                for (int i = q.front; i <= q.rear; i++) {
                    free(q.Q[i].caller);
                }
                free(q.Q);
                exit(0);
            default:
                printf("Invalid choice.\n");
        }
    }
    return 0;
}

// Create the queue
void createQueue(struct Queue *q, int size) {
    q->size = size;
    q->front = -1;
    q->rear = -1;
    q->Q = (struct Call *)malloc(size * sizeof(struct Call));
    if (q->Q == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
}

// Check if the queue is full
int isFull(struct Queue *q) {
    return q->rear == q->size - 1;
}

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

// Enqueue a call
void enQueue(struct Queue *q, int id, char *caller) {
    if (isFull(q)) {
        printf("Queue is full. Cannot enqueue.\n");
        return;
    }
    if (q->front == -1) {
        q->front = 0;
    }
    q->rear++;
    q->Q[q->rear].id = id;

    // Allocate memory for caller name
    q->Q[q->rear].caller = (char *)malloc((strlen(caller) + 1) * sizeof(char));
    if (q->Q[q->rear].caller == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    strcpy(q->Q[q->rear].caller, caller);
}

// Dequeue a call
struct Call deQueue(struct Queue *q) {
    struct Call dequeued = {0, NULL};
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return dequeued;
    }
    dequeued = q->Q[q->front];
    q->front++;
    if (q->front > q->rear) {
        q->front = -1;
        q->rear = -1;
    }
    return dequeued;
}

// Display the queue
void displayQueue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Current Calls in the Queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("Call ID: %d, Caller: %s\n", q->Q[i].id, q->Q[i].caller);
    }
}
```

```
PS C:\Users\betti\Desktop\Training\Day23> ./task6

Enter the total number of incoming calls: 2


1. Add Call

2. Remove Call

3. View Calls

4. Exit

Enter your choice: 1

Enter the call ID: 101

Enter the caller's name: Betti

Call added to the queue.


1. Add Call

2. Remove Call

3. View Calls
```

```
4. Exit

Enter your choice: 1

Enter the call ID: 102

Enter the caller's name: Akash

Call added to the queue.


1. Add Call

2. Remove Call

3. View Calls

4. Exit

Enter your choice: 2

Dequeued Call - ID: 101, Caller: Betti


1. Add Call

2. Remove Call

3. View Calls

4. Exit

Enter your choice: 3

Current Calls in the Queue:

Call ID: 102, Caller: Akash


1. Add Call

2. Remove Call

3. View Calls

4. Exit

Enter your choice: 4
```

7.

```c
/*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>

// Print job structure
struct Print {
    int reqID;
    char *userID;
};

// Queue structure
struct Queue {
    int size;
    int front;
    int rear;
    struct Print *Q;
};

// Global Queue
struct Queue q;

// Function prototypes
void createQueue(struct Queue *, int);
int isFull(struct Queue *);
int isEmpty(struct Queue *);
void enQueue(struct Queue *, int, char *);
struct Print deQueue(struct Queue *, int);
void displayQueue(struct Queue *);

int main() {
    int size;
    printf("Enter the maximum no:of print requests to handle: ");
    scanf("%d", &size);
    createQueue(&q, size);

    int choice;
    while (1) {
        printf("\n1. Add a new print job\n2. Cancel a specific job\n3. Print 
jobs\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                if (isFull(&q)) {
                    printf("Queue is full.\n");
                } else {
                    int id;
                    char user[50];
                    printf("Enter the request ID: ");
                    scanf("%d", &id);
                    printf("Enter the user's ID: ");
                    scanf(" %[^\n]", user); // Take input including spaces
                    enQueue(&q, id, user);
                    printf("Job added to the queue.\n");
                }
                break;
            case 2:
                if (isEmpty(&q)) {
                    printf("Queue is empty.\n");
                } else {
                    int cancelID;
                    printf("Enter the ID of the job to cancel: ");
                    scanf("%d", &cancelID);
                    struct Print dequeued = deQueue(&q, cancelID);
                    if (dequeued.reqID != 0) {
                        printf("Dequeued Job - ID: %d, User: %s\n", 
dequeued.reqID, dequeued.userID);
                        free(dequeued.userID); // Free memory for dequeued user
                    }
                }
                break;
            case 3:
                displayQueue(&q);
                break;
            case 4:
                // Free allocated memory for all remaining jobs
                for (int i = q.front; i <= q.rear; i++) {
                    free(q.Q[i].userID);
                }
                free(q.Q);
                exit(0);
            default:
                printf("Invalid choice.\n");
        }
    }
    return 0;
}

// Create the queue
void createQueue(struct Queue *q, int size) {
    q->size = size;
    q->front = -1;
    q->rear = -1;
    q->Q = (struct Print *)malloc(size * sizeof(struct Print));
    if (q->Q == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
}

// Check if the queue is full
int isFull(struct Queue *q) {
    return q->rear == q->size - 1;
}

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

// Enqueue a print job
void enQueue(struct Queue *q, int id, char *user) {
    if (isFull(q)) {
        printf("Queue is full. Cannot enqueue.\n");
        return;
    }
    if (q->front == -1) {
        q->front = 0;
    }
    q->rear++;
    q->Q[q->rear].reqID = id;

    // Allocate memory for user ID and copy it
    q->Q[q->rear].userID = (char *)malloc((strlen(user) + 1) * sizeof(char));
    if (q->Q[q->rear].userID == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    strcpy(q->Q[q->rear].userID, user);
}

// Dequeue all jobs up to a specific job, including that job
struct Print deQueue(struct Queue *q, int cancelID) {
    struct Print dequeued = {0, NULL};
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return dequeued;
    }

    // Search for the job to cancel
    for (int i = q->front; i <= q->rear; i++) {
        if (q->Q[i].reqID == cancelID) {
            // Extract job details before freeing memory
            dequeued.reqID = q->Q[i].reqID;
            dequeued.userID = (char *)malloc((strlen(q->Q[i].userID) + 1) *
sizeof(char));
            if (dequeued.userID == NULL) {
                printf("Memory allocation failed.\n");
                exit(1);
            }
            strcpy(dequeued.userID, q->Q[i].userID);

            // Free memory for all cancelled jobs up to this point
            for (int j = q->front; j <= i; j++) {
                free(q->Q[j].userID);
            }

            // Update the front pointer
            q->front = i + 1;

            // Reset the queue if all jobs are cancelled
            if (q->front > q->rear) {
                q->front = -1;
                q->rear = -1;
            }

            return dequeued;
```



```c
        }
    }

    // If the cancelID is not found
    printf("Job ID %d not found in the queue.\n", cancelID);
    return dequeued;
}

// Display the queue
void displayQueue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Current Print Jobs in the Queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("Request ID: %d, User ID: %s\n", q->Q[i].reqID, q->Q[i].userID);
    }
}
```

```
PS C:\Users\betti\Desktop\Training\Day23> ./task7

Enter the maximum no:of print requests to handle: 3


1. Add a new print job

2. Cancel a specific job

3. Print jobs

4. Exit

Enter your choice: 1

Enter the request ID: 101

Enter the user's ID: Alice

Job added to the queue.


1. Add a new print job

2. Cancel a specific job

3. Print jobs

4. Exit
```

```
Enter your choice: 1

Enter the request ID: 102

Enter the user's ID: Bob

Job added to the queue.


1. Add a new print job

2. Cancel a specific job

3. Print jobs

4. Exit

Enter your choice: 1

Enter the request ID: 103

Enter the user's ID: Charlie

Job added to the queue.


1. Add a new print job

2. Cancel a specific job

3. Print jobs

4. Exit

Enter your choice: 3

Current Print Jobs in the Queue:

Request ID: 101, User ID: Alice

Request ID: 102, User ID: Bob

Request ID: 103, User ID: Charlie


1. Add a new print job

2. Cancel a specific job

3. Print jobs

4. Exit
```

```
Enter your choice: 2

Enter the ID of the job to cancel: 102

Dequeued Job - ID: 102, User: Bob


1. Add a new print job

2. Cancel a specific job

3. Print jobs

4. Exit

Enter your choice: 3

Current Print Jobs in the Queue:

Request ID: 103, User ID: Charlie


1. Add a new print job

2. Cancel a specific job

3. Print jobs

4. Exit

Enter your choice: 4
```

8.

```c
/*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.*/

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

// Structure for a ticket holder
struct Ticket {
    int id;
    char *name;
};
```

```c
// Queue structure
struct Queue {
    int size;
    int front;
    int rear;
    struct Ticket *Q;
};

// Function prototypes
void createQueue(struct Queue *, int);
int isFull(struct Queue *);
int isEmpty(struct Queue *);
void enQueue(struct Queue *, int, char *);
struct Ticket deQueue(struct Queue *);
void displayQueue(struct Queue *);

// Main function
int main() {
    int size;
    printf("Enter the maximum number of people in the ticket queue: ");
    scanf("%d", &size);
    struct Queue q;
    createQueue(&q, size);

    int choice;
    while (1) {
        printf("\n1. Join the queue\n2. Buy ticket\n3. View queue\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                if (isFull(&q)) {
                    printf("The queue is full. No more people can join.\n");
                } else {
                    int id;
                    char name[50];
                    printf("Enter your ID: ");
                    scanf("%d", &id);
                    printf("Enter your name: ");
                    scanf(" %[^\n]", name);
                    enQueue(&q, id, name);
                    printf("You have joined the queue.\n");
                }
                break;
            case 2:
                if (isEmpty(&q)) {
                    printf("The queue is empty. No tickets to process.\n");
                } else {
                    struct Ticket t = deQueue(&q);
                    printf("Ticket processed for ID: %d, Name: %s\n", t.id,
t.name);
                    free(t.name);
                }
                break;
            case 3:
                displayQueue(&q);
                break;
            case 4:
                // Free allocated memory
                for (int i = q.front; i <= q.rear; i++) {
                    free(q.Q[i].name);
                }
                free(q.Q);
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}

// Create the queue
void createQueue(struct Queue *q, int size) {
    q->size = size;
    q->front = -1;
    q->rear = -1;
    q->Q = (struct Ticket *)malloc(size * sizeof(struct Ticket));
    if (q->Q == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
}

// Check if the queue is full
```

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

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

// Add a person to the queue
void enQueue(struct Queue *q, int id, char *name) {
    if (isFull(q)) {
        printf("Queue is full. Cannot add more people.\n");
        return;
    }
    if (q->front == -1) {
        q->front = 0;
    }
    q->rear++;
    q->Q[q->rear].id = id;

    // Allocate memory for the person's name
    q->Q[q->rear].name = (char *)malloc((strlen(name) + 1) * sizeof(char));
    if (q->Q[q->rear].name == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    strcpy(q->Q[q->rear].name, name);
}

// Remove a person from the queue
struct Ticket deQueue(struct Queue *q) {
    struct Ticket ticket = {0, NULL};
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return ticket;
    }
    ticket = q->Q[q->front];
    q->front++;
    if (q->front > q->rear) {
        q->front = -1;
        q->rear = -1;
    }
    return ticket;
}

// Display the queue
void displayQueue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("The queue is empty.\n");
        return;
    }
    printf("Current people in the queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("ID: %d, Name: %s\n", q->Q[i].id, q->Q[i].name);
    }
}
```

```
PS C:\Users\betti\Desktop\Training\Day23> ./task8

Enter the maximum number of people in the ticket queue: 3


1. Join the queue

2. Buy ticket

3. View queue

4. Exit

Enter your choice: 1

Enter your ID: 101

Enter your name: Alice

You have joined the queue.


1. Join the queue

2. Buy ticket

3. View queue

4. Exit

Enter your choice: 1

Enter your ID: 102

Enter your name: Bob
```

```
You have joined the queue.

1. Join the queue
2. Buy ticket
3. View queue
4. Exit
Enter your choice: 3
Current people in the queue:
ID: 101, Name: Alice
ID: 102, Name: Bob

1. Join the queue
2. Buy ticket
3. View queue
4. Exit
Enter your choice: 2
Ticket processed for ID: 101, Name: Alice

1. Join the queue
2. Buy ticket
3. View queue
4. Exit
Enter your choice: 3
Current people in the queue:
ID: 102, Name: Bob

1. Join the queue
2. Buy ticket
```

```
3. View queue

4. Exit

Enter your choice: 4
```