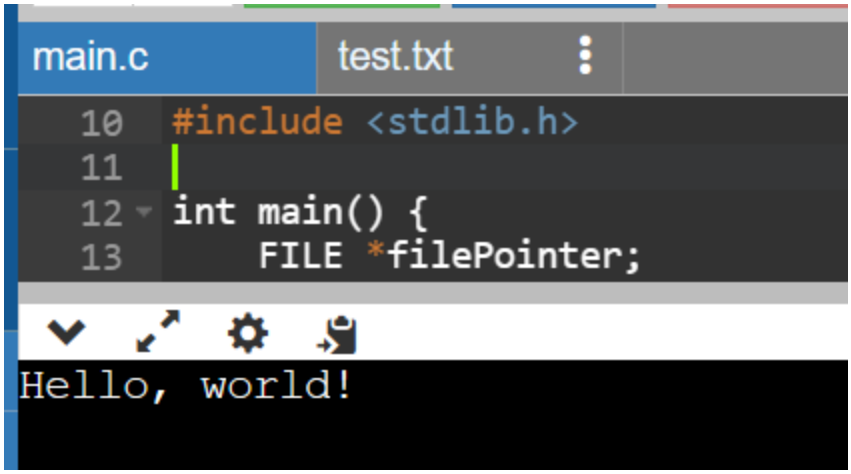
1. a, b, c.

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

int main() {
    FILE *filePointer;
    char buffer[100];

    // (a) Create and write to a file
    // Open the file in write mode
    filePointer = fopen("test.txt", "w");
    if (filePointer == NULL) {
        perror("Error opening file for writing");
        return EXIT_FAILURE;
    }

    // Write "Hello, world!" to the file
    fprintf(filePointer, "Hello, world!\n");

    // Close the file
    fclose(filePointer);

    // (b) Read from the file
    // Open the file in read mode
    filePointer = fopen("test.txt", "r");
    if (filePointer == NULL) {
        perror("Error opening file for reading");
        return EXIT_FAILURE;
    }
```

```c
    // Read the contents of the file into a buffer
    if (fgets(buffer, sizeof(buffer), filePointer) == NULL) {
        perror("Error reading from file");
        // Close the file before returning
        fclose(filePointer);
        return EXIT_FAILURE;
    }

    // Print the contents of the buffer to the console
    printf("%s", buffer);

    // Close the file
    fclose(filePointer);

    return EXIT_SUCCESS;
}
```



2. .

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

// (a) Define a C structure to model a student record
typedef struct {
    int ID;
    char name[50];
    float GPA;
} Student;

int main() {
    // (b) Initialize an array of 'Student' records
    Student students[] = {
        {1, "Alice Johnson", 3.5},
        {2, "Bob Smith", 3.7},
        {3, "Charlie Davis", 3.9}
    };
    int numberOfStudents = sizeof(students) / sizeof(students[0]);

    // Open a file in write mode
    FILE *filePointer = fopen("students.dat", "wb");
    if (filePointer == NULL) {
        perror("Error opening file for writing");
        return EXIT_FAILURE;
    }

    // Write the array to the file
    if (fwrite(students, sizeof(Student), numberOfStudents, filePointer) != numberOfStudents) {
        perror("Error writing to file");
        fclose(filePointer);
        return EXIT_FAILURE;
    }

```

```c
    // Close the file after writing
    fclose(filePointer);

    // (c) Read back the structured data
    Student readStudents[numberOfStudents];

    // Open the previously written file in read mode
    filePointer = fopen("students.dat", "rb");
    if (filePointer == NULL) {
        perror("Error opening file for reading");
        return EXIT_FAILURE;
    }

    // Read the student records into a new array
    if (fread(readStudents, sizeof(Student), numberOfStudents, filePointer) != numberOfStudents) {
        perror("Error reading from file");
        fclose(filePointer);
        return EXIT_FAILURE;
    }

    // Iterate over the array and print out each student record
    for (int i = 0; i < numberOfStudents; i++) {
        printf("ID: %d, Name: %s, GPA: %.2f\n", readStudents[i].ID, readStudents[i].name, readStudents[i].GPA);
    }

    // Close the file after reading the contents
    fclose(filePointer);

    return EXIT_SUCCESS;
}
```

main.c test.txt students.dat

1 ····Alice Johnson·····`@····Bob Smith·····ÍÍl@····Charlie

input

```
ID: 1, Name: Alice Johnson, GPA: 3.50
ID: 2, Name: Bob Smith, GPA: 3.70
ID: 3, Name: Charlie Davis, GPA: 3.90
```

3. .

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

#define MAX_SIZE 5 // Define the maximum size of the stack

// (a) Define a Stack data structure
typedef struct {
    int items[MAX_SIZE];
    int top;
} Stack;

// Function to initialize the stack
void initStack(Stack *s) {
    s->top = -1;
}

// (b) Implement stack operations
// push operation
void push(Stack *s, int value) {
    if (s->top == MAX_SIZE - 1) {
        printf("Stack overflow. Cannot push %d\n", value);
    } else {
        s->top++;
        s->items[s->top] = value;
        printf("Pushed %d to the stack\n", value);
    }
}

// pop operation
int pop(Stack *s) {
    if (s->top == -1) {
        printf("Stack underflow. Cannot pop\n");
        return -1; // Return -1 to indicate error
    } else {
        int value = s->items[s->top];
```

```
43          int value = s->items[s->top];
44          s->top--;
45          printf("Popped %d from the stack\n", value);
46          return value;
47      }
48  }
49
50  // peek operation
51  int peek(Stack *s) {
52      if (s->top == -1) {
53          printf("Stack is empty. Cannot peek\n");
54          return -1; // Return -1 to indicate error
55      } else {
56          printf("Top element is %d\n", s->items[s->top]);
57          return s->items[s->top];
58      }
59  }
60
61  // (c) Test suite for stack functionality
62  int main() {
63      Stack s;
64      initStack(&s);
65
66      // Perform a sequence of push operations
67      push(&s, 10);
68      peek(&s); // Should show 10
69      push(&s, 20);
70      peek(&s); // Should show 20
71      push(&s, 30);
72      peek(&s); // Should show 30
73      push(&s, 40);
74      peek(&s); // Should show 40
75      push(&s, 50);
```

```
    push(&s, 50);
    peek(&s); // Should show 50
    push(&s, 60); // Should indicate stack overflow

    // Perform a sequence of pop operations
    pop(&s); // Should remove 50
    pop(&s); // Should remove 40
    pop(&s); // Should remove 30
    pop(&s); // Should remove 20
    pop(&s); // Should remove 10
    pop(&s); // Should indicate stack underflow

    return 0;
}
```

input

```
Pushed 10 to the stack
Top element is 10
Pushed 20 to the stack
Top element is 20
Pushed 30 to the stack
Top element is 30
Pushed 40 to the stack
Top element is 40
Pushed 50 to the stack
Top element is 50
Stack overflow. Cannot push 60
Popped 50 from the stack
Popped 40 from the stack
Popped 30 from the stack
Popped 20 from the stack
Popped 10 from the stack
Stack underflow. Cannot pop
```

4. .

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

#define MAX_SIZE 5 // Define the maximum size of the queue

// (a) Define the Queue data structure
typedef struct {
    int items[MAX_SIZE];
    int front, rear;
} Queue;

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

// Check if the queue is full
int isFull(Queue *q) {
    if ((q->rear + 1) % MAX_SIZE == q->front) {
        return 1; // True
    } else {
        return 0; // False
    }
}

// Check if the queue is empty
int isEmpty(Queue *q) {
    if (q->front == -1) {
        return 1; // True
    } else {
        return 0; // False
    }
}
```

```c
}

// (b) Implement queue operations
// enqueue operation
void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue overflow. Cannot enqueue %d\n", value);
    } else {
        if (isEmpty(q)) {
            q->front = 0;
        }
        q->rear = (q->rear + 1) % MAX_SIZE;
        q->items[q->rear] = value;
        printf("Enqueued %d\n", value);
    }
}

// dequeue operation
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue underflow. Cannot dequeue\n");
        return -1; // Return -1 to indicate error
    } else {
        int value = q->items[q->front];
        if (q->front == q->rear) { // Queue becomes empty
            q->front = -1;
            q->rear = -1;
        } else {
            q->front = (q->front + 1) % MAX_SIZE;
        }
        printf("Dequeued %d\n", value);
        return value;
    }
}

```

```c
// front operation
int front(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot retrieve front\n");
        return -1; // Return -1 to indicate error
    } else {
        printf("Front element is %d\n", q->items[q->front]);
        return q->items[q->front];
    }
}

// (c) Test sequence for the queue operations
int main() {
    Queue q;
    initQueue(&q);

    // Enqueue operations
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    enqueue(&q, 40);
    enqueue(&q, 50);
    enqueue(&q, 60); // Should indicate queue overflow

    // Peek at the front value
    front(&q);

    // Dequeue operations
    dequeue(&q);
    dequeue(&q);
    dequeue(&q);
    dequeue(&q);
    dequeue(&q);
    dequeue(&q); // Should indicate queue underflow
```

```
110     dequeue(&q); // Should indicate queue underflow
111
112     // Peek at the front value again
113     front(&q);
114
115     return 0;
116 }
117
```

```
Enqueued 10
Enqueued 20
Enqueued 30
Enqueued 40
Enqueued 50
Queue overflow. Cannot enqueue 60
Front element is 10
Dequeued 10
Dequeued 20
Dequeued 30
Dequeued 40
Dequeued 50
Queue underflow. Cannot dequeue
Queue is empty. Cannot retrieve front
```

5. .Continued from line 76 of no 4

```c
// Function to save the state of the queue to a file
void saveQueue(Queue *q, const char *filename) {
    FILE *file = fopen(filename, "wb");
    if (!file) {
        perror("Failed to open file for writing");
        return;
    }
    if (fwrite(q, sizeof(Queue), 1, file) != 1) {
        perror("Failed to write queue to file");
    }
    fclose(file);
    printf("Queue saved to %s\n", filename);
}

// Function to load the state of the queue from a file
void loadQueue(Queue *q, const char *filename) {
    FILE *file = fopen(filename, "rb");
    if (!file) {
        perror("Failed to open file for reading");
        return;
    }
    if (fread(q, sizeof(Queue), 1, file) != 1) {
        perror("Failed to read queue from file");
    }
    fclose(file);
    printf("Queue loaded from %s\n", filename);
}

// (b) Test scenario for data persistence
int main() {
    Queue q;
    initQueue(&q);
```

main.c test.txt students.dat queue.dat

```
107     initQueue(&q);
108
109     // Enqueue operations
110     enqueue(&q, 10);
111     enqueue(&q, 20);
112     enqueue(&q, 30);
113
114     // Save the current state of the queue to a file
115     saveQueue(&q, "queue.dat");
116
117     // Clear the queue
118     initQueue(&q);
119
120     // Load the queue's state from the file
121     loadQueue(&q, "queue.dat");
122
123     // Validate the restored queue
124     dequeue(&q); // Should dequeue 10
125     dequeue(&q); // Should dequeue 20
126     dequeue(&q); // Should dequeue 30
127
128     return 0;
129 }
```

inp

```
Enqueued 10
Enqueued 20
Enqueued 30
Queue saved to queue.dat
Queue loaded from queue.dat
Dequeued 10
Dequeued 20
Dequeued 30
```

6. .

```c
// (a) Update stack operations for Student data type
// push operation
void push(Stack *s, Student student) {
    if (isFull(s)) {
        printf("Stack overflow. Cannot push new record.\n");
    } else {
        s->top++;
        s->items[s->top] = student;
        printf("Pushed: ID=%d, Name=%s, GPA=%.2f\n", student.ID, student.name, student.GPA);
    }
}

// pop operation
Student pop(Stack *s) {
    Student emptyStudent = {0, "", 0.0};
    if (isEmpty(s)) {
        printf("Stack underflow. Cannot pop.\n");
        return emptyStudent;
    } else {
        Student student = s->items[s->top];
        s->top--;
        return student;
    }
}

// peek operation
Student peek(Stack *s) {
    Student emptyStudent = {0, "", 0.0};
    if (isEmpty(s)) {
        printf("Stack is empty.\n");
        return emptyStudent;
    } else {
        return s->items[s->top];
    }
}
```

```c
}

// (b) Implement file I/O operations for the stack
void saveStack(Stack *s, const char *filename) {
    FILE *file = fopen(filename, "wb");
    if (!file) {
        perror("Failed to open file for writing");
        return;
    }
    for (int i = 0; i <= s->top; i++) {
        fwrite(&s->items[i], sizeof(Student), 1, file);
    }
    fclose(file);
    printf("Stack saved to %s\n", filename);
}

void loadStack(Stack *s, const char *filename) {
    FILE *file = fopen(filename, "rb");
    if (!file) {
        perror("Failed to open file for reading");
        return;
    }
    initStack(s); // Clear current stack
    Student temp;
    while (fread(&temp, sizeof(Student), 1, file)) {
        push(s, temp); // Push each student onto the stack
    }
    fclose(file);
    printf("Stack loaded from %s\n", filename);
}

// (c) Design a test sequence for the stack operations
int main() {
    Stack s;
    initStack(&s);
```

```c
    initStack(&s);

    // Populate the stack with Student records
    push(&s, (Student){1, "Alice Johnson", 3.5});
    push(&s, (Student){2, "Bob Smith", 3.7});
    push(&s, (Student){3, "Charlie Davis", 3.9});

    // Save the stack to a file
    saveStack(&s, "stack.dat");

    // Clear the stack in memory
    initStack(&s);

    // Reload the stack from the file
    loadStack(&s, "stack.dat");

    // Verify stack order and data integrity
    Student temp = pop(&s);
    while (temp.ID != 0) {
        printf("Popped: ID=%d, Name=%s, GPA=%.2f\n", temp.ID, temp.name, temp.GPA);
        temp = pop(&s);
    }

    return 0;
}
```

main.c | test.txt | students.dat | queue.dat | stack.dat

```
1  ····Alice Johnson····`@····Bob Smith··
```

input

```
Pushed: ID=1, Name=Alice Johnson, GPA=3.50
Pushed: ID=2, Name=Bob Smith, GPA=3.70
Pushed: ID=3, Name=Charlie Davis, GPA=3.90
Stack saved to stack.dat
Pushed: ID=1, Name=Alice Johnson, GPA=3.50
Pushed: ID=2, Name=Bob Smith, GPA=3.70
Pushed: ID=3, Name=Charlie Davis, GPA=3.90
Stack loaded from stack.dat
Popped: ID=3, Name=Charlie Davis, GPA=3.90
Popped: ID=2, Name=Bob Smith, GPA=3.70
Popped: ID=1, Name=Alice Johnson, GPA=3.50
Stack underflow. Cannot pop.
```