

1. a, b, c.

```
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 int main() {
13     FILE *filePointer;
14     char buffer[100];
15
16     // (a) Create and write to a file
17     // Open the file in write mode
18     filePointer = fopen("test.txt", "w");
19     if (filePointer == NULL) {
20         perror("Error opening file for writing");
21         return EXIT_FAILURE;
22     }
23
24     // Write "Hello, world!" to the file
25     fprintf(filePointer, "Hello, world!\n");
26
27     // Close the file
28     fclose(filePointer);
29
30     // (b) Read from the file
31     // Open the file in read mode
32     filePointer = fopen("test.txt", "r");
33     if (filePointer == NULL) {
34         perror("Error opening file for reading");
35         return EXIT_FAILURE;
36     }
37
38     // Read the contents of the file into a buffer
39     if (fgets(buffer, sizeof(buffer), filePointer) == NULL) {
40         perror("Error reading from file");
41         // Close the file before returning
42         fclose(filePointer);
43         return EXIT_FAILURE;
44     }
45
46     // Print the contents of the buffer to the console
47     printf("%s", buffer);
48
49     // Close the file
50     fclose(filePointer);
51
52     return EXIT_SUCCESS;
53 }
```

The screenshot shows a terminal window with two tabs: "main.c" and "test.txt". The "main.c" tab contains the following C code:

```
10 #include <stdlib.h>
11 |
12 int main() {
13     FILE *filePointer;
```

The "test.txt" tab shows the output of the program: "Hello, world!".

2. .

```
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 // (a) Define a C structure to model a student record
13 typedef struct {
14     int ID;
15     char name[50];
16     float GPA;
17 } Student;
18
19 int main() {
20     // (b) Initialize an array of 'Student' records
21     Student students[] = [
22         {1, "Alice Johnson", 3.5},
23         {2, "Bob Smith", 3.7},
24         {3, "Charlie Davis", 3.9}
25     ];
26     int numberOfStudents = sizeof(students) / sizeof(students[0]);
27
28     // Open a file in write mode
29     FILE *filePointer = fopen("students.dat", "wb");
30     if (filePointer == NULL) {
31         perror("Error opening file for writing");
32         return EXIT_FAILURE;
33     }
34
35     // Write the array to the file
36     if (fwrite(students, sizeof(Student), numberOfStudents, filePointer) != numberOfStudents) {
37         perror("Error writing to file");
38         fclose(filePointer);
39         return EXIT_FAILURE;
40     }
41 }
```

```
41     // Close the file after writing
42     fclose(filePointer);
43
44     // (c) Read back the structured data
45     Student readStudents[numberOfStudents];
46
47     // Open the previously written file in read mode
48     filePointer = fopen("students.dat", "rb");
49     if (filePointer == NULL) {
50         perror("Error opening file for reading");
51         return EXIT_FAILURE;
52     }
53
54     // Read the student records into a new array
55     if (fread(readStudents, sizeof(Student), numberOfStudents, filePointer) != numberOfStudents) {
56         perror("Error reading from file");
57         fclose(filePointer);
58         return EXIT_FAILURE;
59     }
60
61     // Iterate over the array and print out each student record
62     for (int i = 0; i < numberOfStudents; i++) {
63         printf("ID: %d, Name: %s, GPA: %.2f\n", readStudents[i].ID, readStudents[i].name, readStudents[i].GPA);
64     }
65
66     // Close the file after reading the contents
67     fclose(filePointer);
68
69
70     return EXIT_SUCCESS;
71 }
```

```
main.c    test.txt    students.dat
1 Alice Johnson @ Bob Smith Charlie Davis
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.

```
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 #define MAX_SIZE 5 // Define the maximum size of the stack
13
14 // (a) Define a Stack data structure
15 typedef struct {
16     int items[MAX_SIZE];
17     int top;
18 } Stack;
19
20 // Function to initialize the stack
21 void initStack(Stack *s) {
22     s->top = -1;
23 }
24
25 // (b) Implement stack operations
26 // push operation
27 void push(Stack *s, int value) {
28     if (s->top == MAX_SIZE - 1) {
29         printf("Stack overflow. Cannot push %d\n", value);
30     } else {
31         s->top++;
32         s->items[s->top] = value;
33         printf("Pushed %d to the stack\n", value);
34     }
35 }
36
37 // pop operation
38 int pop(Stack *s) {
39     if (s->top == -1) {
40         printf("Stack underflow. Cannot pop\n");
41         return -1; // Return -1 to indicate error
42     } else {
43         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);
```

```
75     push(&s, 50);
76     peek(&s); // Should show 50
77     push(&s, 60); // Should indicate stack overflow
78
79     // Perform a sequence of pop operations
80     pop(&s); // Should remove 50
81     pop(&s); // Should remove 40
82     pop(&s); // Should remove 30
83     pop(&s); // Should remove 20
84     pop(&s); // Should remove 10
85     pop(&s); // Should indicate stack underflow
86
87     return 0;
88 }
```

```
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. .

```
9 #include <stdio.h>
10 #include <stdlib.h>
11
12 #define MAX_SIZE 5 // Define the maximum size of the queue
13
14 // (a) Define the Queue data structure
15 typedef struct {
16     int items[MAX_SIZE];
17     int front, rear;
18 } Queue;
19
20 // Function to initialize the queue
21 void initQueue(Queue *q) {
22     q->front = -1;
23     q->rear = -1;
24 }
25
26 // Check if the queue is full
27 int isFull(Queue *q) {
28     if ((q->rear + 1) % MAX_SIZE == q->front) {
29         return 1; // True
30     } else {
31         return 0; // False
32     }
33 }
34
35 // Check if the queue is empty
36 int isEmpty(Queue *q) {
37     if (q->front == -1) {
38         return 1; // True
39     } else {
40         return 0; // False
41     }
42 }
```

```
42 }
43
44 // (b) Implement queue operations
45 // enqueue operation
46 void enqueue(Queue *q, int value) {
47     if (isFull(q)) {
48         printf("Queue overflow. Cannot enqueue %d\n", value);
49     } else {
50         if (isEmpty(q)) {
51             q->front = 0;
52         }
53         q->rear = (q->rear + 1) % MAX_SIZE;
54         q->items[q->rear] = value;
55         printf("Enqueued %d\n", value);
56     }
57 }
58
59 // dequeue operation
60 int dequeue(Queue *q) {
61     if (isEmpty(q)) {
62         printf("Queue underflow. Cannot dequeue\n");
63         return -1; // Return -1 to indicate error
64     } else {
65         int value = q->items[q->front];
66         if (q->front == q->rear) { // Queue becomes empty
67             q->front = -1;
68             q->rear = -1;
69         } else {
70             q->front = (q->front + 1) % MAX_SIZE;
71         }
72         printf("Dequeued %d\n", value);
73         return value;
74     }
75 }
76
```

```
77 // front operation
78 int front(Queue *q) {
79     if (isEmpty(q)) {
80         printf("Queue is empty. Cannot retrieve front\n");
81         return -1; // Return -1 to indicate error
82     } else {
83         printf("Front element is %d\n", q->items[q->front]);
84         return q->items[q->front];
85     }
86 }
87
88 // (c) Test sequence for the queue operations
89 int main() {
90     Queue q;
91     initQueue(&q);
92
93     // Enqueue operations
94     enqueue(&q, 10);
95     enqueue(&q, 20);
96     enqueue(&q, 30);
97     enqueue(&q, 40);
98     enqueue(&q, 50);
99     enqueue(&q, 60); // Should indicate queue overflow
100
101    // Peek at the front value
102    front(&q);
103
104    // Dequeue operations
105    dequeue(&q);
106    dequeue(&q);
107    dequeue(&q);
108    dequeue(&q);
109    dequeue(&q);
110    dequeue(&q); // Should indicate queue underflow
111
```

```
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

```
76 // Function to save the state of the queue to a file
77 void saveQueue(Queue *q, const char *filename) {
78     FILE *file = fopen(filename, "wb");
79     if (!file) {
80         perror("Failed to open file for writing");
81         return;
82     }
83     if (fwrite(q, sizeof(Queue), 1, file) != 1) {
84         perror("Failed to write queue to file");
85     }
86     fclose(file);
87     printf("Queue saved to %s\n", filename);
88 }
89
90 // Function to load the state of the queue from a file
91 void loadQueue(Queue *q, const char *filename) {
92     FILE *file = fopen(filename, "rb");
93     if (!file) {
94         perror("Failed to open file for reading");
95         return;
96     }
97     if (fread(q, sizeof(Queue), 1, file) != 1) {
98         perror("Failed to read queue from file");
99     }
100    fclose(file);
101    printf("Queue loaded from %s\n", filename);
102 }
103
104 // (b) Test scenario for data persistence
105 int main() {
106     Queue q;
107     initQueue(&q);
108 }
```

The screenshot shows a terminal window with the following content:

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

6.

```
43 // (a) Update stack operations for Student data type
44 // push operation
45 void push(Stack *s, Student student) {
46     if (isFull(s)) {
47         printf("Stack overflow. Cannot push new record.\n");
48     } else {
49         s->top++;
50         s->items[s->top] = student;
51         printf("Pushed: ID=%d, Name=%s, GPA=%.2f\n", student.ID, student.name, student.GPA);
52     }
53 }
54
55 // pop operation
56 Student pop(Stack *s) {
57     Student emptyStudent = {0, "", 0.0};
58     if (isEmpty(s)) {
59         printf("Stack underflow. Cannot pop.\n");
60         return emptyStudent;
61     } else {
62         Student student = s->items[s->top];
63         s->top--;
64         return student;
65     }
66 }
67
68 // peek operation
69 Student peek(Stack *s) {
70     Student emptyStudent = {0, "", 0.0};
71     if (isEmpty(s)) {
72         printf("Stack is empty.\n");
73         return emptyStudent;
74     } else {
75         return s->items[s->top];
76     }
77 }
```

```
76     }
77 }
78
79 // (b) Implement file I/O operations for the stack
80 void saveStack(Stack *s, const char *filename) {
81     FILE *file = fopen(filename, "wb");
82     if (!file) {
83         perror("Failed to open file for writing");
84         return;
85     }
86     for (int i = 0; i <= s->top; i++) {
87         fwrite(&s->items[i], sizeof(Student), 1, file);
88     }
89     fclose(file);
90     printf("Stack saved to %s\n", filename);
91 }
92
93 void loadStack(Stack *s, const char *filename) {
94     FILE *file = fopen(filename, "rb");
95     if (!file) {
96         perror("Failed to open file for reading");
97         return;
98     }
99     initStack(s); // Clear current stack
100    Student temp;
101    while (fread(&temp, sizeof(Student), 1, file)) {
102        push(s, temp); // Push each student onto the stack
103    }
104    fclose(file);
105    printf("Stack loaded from %s\n", filename);
106 }
107
108 // (c) Design a test sequence for the stack operations
109 int main() {
110     Stack s;
111     initStack(&s);
112 }
```

```
110     stack_s;
111     initStack(&s);
112
113     // Populate the stack with Student records
114     push(&s, (Student){1, "Alice Johnson", 3.5});
115     push(&s, (Student){2, "Bob Smith", 3.7});
116     push(&s, (Student){3, "Charlie Davis", 3.9});
117
118     // Save the stack to a file
119     saveStack(&s, "stack.dat");
120
121     // Clear the stack in memory
122     initStack(&s);
123
124     // Reload the stack from the file
125     loadStack(&s, "stack.dat");
126
127     // Verify stack order and data integrity
128     Student temp = pop(&s);
129     while (temp.ID != 0) {
130         printf("Popped: ID=%d, Name=%s, GPA=%.2f\n", temp.ID, temp.name, temp.GPA);
131         temp = pop(&s);
132     }
133
134     return 0;
135 }
136
```

```
main.c test.txt students.dat queue.dat stack.dat
1 Alice Johnson @ Bob Smith
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.
```