1. **Explain the concept of pointers in C and write a program to swap the values of two variables using pointers.**

   Pointers in C are variables that store the address of another variable. They are declared using the asterisk **(*)** symbol. For example, an integer pointer is written as **int *a**.

```
#include <stdio.h>

void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int main() {
    int x = 10;
    int y = 20;

    printf("Before swap: x = %d, y = %d\n", x, y);

    swap(&x, &y);

    printf("After swap: x = %d, y = %d\n", x, y);

    return 0;
}
```

2. **Write a C program to reverse a given string without using any additional library functions.**

```
#include <stdio.h>

#include <string.h>

void reverseString(char str[]) {
int length = strlen(str);
char temp[length];
int i, j;

for (i = 0, j = length - 1; i < length; i++, j--) {
temp[i] = str[j];
}

for (i = 0; i < length; i++) {
str[i] = temp[i];
} }

int main() {
char str[] = "Hello, World!";

printf("Original string: %s\n", str);
reverseString(str);
printf("Reversed string: %s\n", str);

return 0;
}
```

3. **Explain the concept of structures in C and write a program to store student information (name, roll number, marks) using a structure.**

In C programming, a structure (often referred to simply as a ‘struct’) is a user-defined data type that allows us to group together different types of variables under a single name. It provides a way to store a collection of heterogeneous data items.

```
#include <stdio.h>

struct Student {
    int rollNumber;
    char name[50];
    float marks;
};

int main() {

    struct Student student1;

    student1.rollNumber = 16;
    student1.name = "Jimmy";
    student1.marks = 90;

    // Accessing and printing the members of 'student1'
    printf("Roll Number: %d\n", student1.rollNumber);
    printf("Name: %s\n", student1.name);
    printf("Marks: %.2f\n", student1.marks);

    return 0;
}
```

4. **Differentiate between single-linked lists and doubly-linked lists in C. Write code snippets to create a node and perform a basic insertion operation in a singly-linked list.**

a) Singly Linked List:

1. **Structure**:
    a) Each node contains data and a pointer/reference to the next node in the sequence.
    b) The last node points to NULL, indicating the end of the list.
2. **Traversal**:
    a) Traversal is possible only in one direction (forward).
    b) To access elements, you start from the head (first node) and move sequentially through each node until you reach the desired node or the end (NULL).
3. **Memory Usage**:
    a) Requires less memory per node compared to a doubly linked list because it stores only one reference (next pointer).

b) Doubly Linked List:

1. **Structure**:
    - Each node contains data and pointers/references to both the next node and the previous node.
    - The first node's previous pointer and the last node's next pointer point to NULL.
2. **Traversal**:
    - Allows traversal in both directions: forward (using the next pointer) and backward (using the previous pointer).
    - This bidirectional traversal facilitates operations that require accessing nodes in both directions.

3. **Memory Usage**:
    - Requires more memory per node compared to a singly linked list because it stores two references (next and previous pointers).
4. **Operations**:
    - Insertions and deletions at the beginning and end of the list are efficient (constant time complexity, O(1)O(1)O(1)), as you have direct access to both the head and tail nodes.
    - Insertions and deletions at any position in the list are also more efficient compared to singly linked lists (constant time if the position is known, otherwise linear time for traversal).