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;

}

1. **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;

}

1. **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.

For ex. **Struct class**{

Int students;

Char Teacher name[30];

}

1. **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.**
2. **Singly Linked List:**
3. **Structure**:
   * Each node contains data and a pointer/reference to the next node in the sequence.
   * The last node points to NULL, indicating the end of the list.
4. **Traversal**:
   * Traversal is possible only in one direction (forward).
   * 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).
5. **Memory Usage**:
   * Requires less memory per node compared to a doubly linked list because it stores only one reference (next pointer).
6. **Doubly Linked List:**
7. **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.
8. **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.
9. **Memory Usage**:
   * Requires more memory per node compared to a singly linked list because it stores two references (next and previous pointers).
10. **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).

**Summary:**

* **Singly Linked List**: Simple structure with each node pointing to the next node. Memory-efficient but limited to forward traversal.
* **Doubly Linked List**: More complex with each node pointing to both the next and previous nodes. Allows bidirectional traversal and more efficient insertions/deletions at any position.