1. Write a program to find the size of a data type without using the `sizeof` operator, use pointer arithmetic.

WTD: Declare a pointer to the given type and increment it. Subtract the original pointer value from the incremented value to get the size.
(e.g.: I/P: int; O/P: 4(based on platform))

```c
#include <stdio.h>

int main() {
    int *ptr1, *ptr2;

    ptr1 = ptr2 + 1; // Increment the pointer by 1

    // Calculate the size of the data type
    int size = (int)ptr1 - (int)ptr2;

    printf("Size of int: %d\n", size);

    return 0;
}
```

2. Design a function that determines if two pointers point to the same array.

WTD: Traverse from both pointers in both forward and backward directions until null or boundary is hit. If both pointers hit the same boundaries, they belong to the same array.
(e.g.: I/P: int arr[] = {1,2,3,4}, *ptr1 = &arr[1], *ptr2 = &arr[3]; O/P: True)

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

// A function that returns true if two pointers point to the same array
bool same_array(int *p1, int *p2) {
    // Initialize two pointers to store the boundaries of the arrays
    int *start1 = p1;
    int *start2 = p2;
    int *end1 = p1;
    int *end2 = p2;
```

```
    // Traverse from p1 in backward direction until null or boundary is
hit
    while (start1 != NULL && *start1 != '\0') {
        start1--;
    }

    // Traverse from p2 in backward direction until null or boundary is
hit
    while (start2 != NULL && *start2 != '\0') {
        start2--;
    }

    // Traverse from p1 in forward direction until null or boundary is hit
    while (end1 != NULL && *end1 != '\0') {
        end1++;
    }

    // Traverse from p2 in forward direction until null or boundary is hit
    while (end2 != NULL && *end2 != '\0') {
        end2++;
    }

    // Compare the boundaries of the arrays
    if (start1 == start2 && end1 == end2) {
        return true; // The pointers belong to the same array
    } else {
        return false; // The pointers belong to different arrays
    }
}

int main() {
    // Test cases
    int arr[] = {1, 2, 3, 4};
    int *ptr1 = &arr[1];
    int *ptr2 = &arr[3];
    printf("%d\n", same_array(ptr1, ptr2)); // Output: 1 (true)

    int arr1[] = {5, 6, 7};
    int arr2[] = {8, 9};
```

```
    int *ptr3 = &arr1[0];
    int *ptr4 = &arr2[0];
    printf("%d\n", same_array(ptr3, ptr4)); // Output: 0 (false)

    return 0;
}
```

3. Create a function that uses pointer arithmetic to count the number of elements in an array without utilizing loop constructs.

WTD: Subtract the pointer to the first element from the pointer just past the last element.
(e.g.: I/P: int arr[] = {1,2,3,4,5} ,O/P: 5)

```
#include <stdio.h>

int countElementsInArray(int arr[], int size) {
    // The size parameter indicates the number of elements in the array
    return size;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int elementCount = countElementsInArray(arr, sizeof(arr) /
sizeof(arr[0]));

    printf("Number of elements in the array: %d\n", elementCount);

    return 0;
}
```

4. Implement a program that swaps two strings using pointers to pointers.

WTD: Use a pointer to pointer to swap the base addresses of the two strings.
(e.g.: I/P: char *str1 = "hello", *str2 = "world" ,O/P: str1 = "world", str2 = "hello")

```c
#include <stdio.h>

void swapStrings(char **str1, char **str2) {
    char *temp = *str1;
    *str1 = *str2;
    *str2 = temp;
}

int main() {
    char *str1 = "hello";
    char *str2 = "world";

    printf("Before Swap:\n");
    printf("str1 = %s\n", str1);
    printf("str2 = %s\n", str2);

    swapStrings(&str1, &str2);

    printf("\nAfter Swap:\n");
    printf("str1 = %s\n", str1);
    printf("str2 = %s\n", str2);

    return 0;
}
```

5. Create a function that segregates even and odd values of an integer array using pointers.

WTD: Use two pointers, one starting from the beginning and the other from the end. Traverse and swap even and odd numbers until the two pointers meet.
(e.g.: I/P: int arr[] = {12,34,9,8,45,90} ,O/P: {12,34,8,90,9,45})

```c
#include <stdio.h>

void segregateEvenOdd(int arr[], int size) {
    int *left = arr;
    int *right = arr + size - 1;

    while (left < right) {
        // Move left pointer to the right while it points to an even number
        while ((*left % 2 == 0) && (left < right)) {
            left++;
        }

        // Move right pointer to the left while it points to an odd number
        while ((*right % 2 != 0) && (left < right)) {
            right--;
        }

        // Swap the elements at left and right pointers
        if (left < right) {
            int temp = *left;
            *left = *right;
            *right = temp;
        }
    }
}

int main() {
    int arr[] = {12, 34, 9, 8, 45, 90};
    int size = sizeof(arr) / sizeof(arr[0]);
```

```
    printf("Original Array:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    segregateEvenOdd(arr, size);

    printf("\nArray After Segregation:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

6. Design a program to concatenate two strings without using standard library functions, only pointers.

WTD: Traverse the first string till the end and then copy the second string from that point.
(e.g.: I/P: char *str1 = "Good ", *str2 = "Morning" ,O/P: "Good Morning")

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

char *concatenateStrings(const char *str1, const char *str2) {
    // Calculate the length of the concatenated string
    int len1 = 0;
    while (str1[len1] != '\0') {
        len1++;
    }

    int len2 = 0;
    while (str2[len2] != '\0') {
        len2++;
    }

    // Allocate memory for the concatenated string
```

```
    char *result = (char *)malloc(len1 + len2 + 1);

    // Copy characters from str1 to result
    int i, j;
    for (i = 0; i < len1; i++) {
        result[i] = str1[i];
    }

    // Copy characters from str2 to result
    for (j = 0; j < len2; j++) {
        result[i + j] = str2[j];
    }

    // Null-terminate the concatenated string
    result[i + j] = '\0';

    return result;
}

int main() {
    const char *str1 = "Good ";
    const char *str2 = "Morning";

    char *concatenated = concatenateStrings(str1, str2);

    printf("Concatenated String: %s\n", concatenated);

    free(concatenated); // Don't forget to free the allocated memory

    return 0;
}
```

7. Implement a function that splits a string into two halves and returns pointers to the beginning of each half.

WTD: Use pointer arithmetic to find the middle of the string. Return the original and the middle pointers.
(e.g.: I/P: "HelloWorld" ,O/P: "Hello", "World")

```c
#include <stdio.h>
#include <stdlib.h> // Include this header for malloc and free
#include <string.h>

void splitString(const char *input, char **firstHalf, char **secondHalf) {
    int len = strlen(input);

    // Calculate the midpoint of the string
    int midpoint = len / 2;

    // Allocate memory for the two halves
    *firstHalf = (char *)malloc(midpoint + 1);
    *secondHalf = (char *)malloc(len - midpoint + 1);

    // Copy the first half of the string
    strncpy(*firstHalf, input, midpoint);
    (*firstHalf)[midpoint] = '\0';

    // Copy the second half of the string
    strcpy(*secondHalf, input + midpoint);
}

int main() {
    const char *input = "HelloWorld";
    char *firstHalf;
    char *secondHalf;

    splitString(input, &firstHalf, &secondHalf);

    printf("First Half: %s\n", firstHalf);
    printf("Second Half: %s\n", secondHalf);

    // Don't forget to free the allocated memory
    free(firstHalf);
    free(secondHalf);

    return 0;
}
```

8. Write a function that trims leading and trailing whitespace from a string using pointers.

WTD: Use two pointers to find the first and last non-whitespace characters. Move characters to trim the string
(e.g.: I/P: " Hello World " ,O/P: "Hello World")

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

void trimString(char *str) {
    if (str == NULL) {
        return; // Handle NULL input gracefully
    }

    char *start = str;
    char *end = str + strlen(str) - 1;

    // Trim leading whitespace
    while (*start == ' ' || *start == '\t' || *start == '\n') {
        start++;
    }

    // Trim trailing whitespace
    while (*end == ' ' || *end == '\t' || *end == '\n') {
        end--;
    }

    // Move characters to the beginning of the string
    while (start <= end) {
        *str = *start;
        str++;
        start++;
    }

    *str = '\0'; // Null-terminate the trimmed string
}
```

```c
int main() {
    char str[] = "  Hello World  ";
    trimString(str);
    printf("Trimmed String: \"%s\"\n", str);

    return 0;
}
```

9. Design a program to find the overlapping part of two arrays using pointers.

WTD: Use two pointers to traverse both arrays. When a common element is found, move both pointers forward.
(e.g.: I/P: int arr1[] = {1,2,3,4,5,6}, arr2[] = {5,6,7,8} ,O/P: {5,6})

```c
#include <stdio.h>

void findOverlap(int arr1[], int size1, int arr2[], int size2) {
    int *ptr1 = arr1;
    int *ptr2 = arr2;

    printf("Overlapping Part: {");

    while (ptr1 < arr1 + size1 && ptr2 < arr2 + size2) {
        if (*ptr1 == *ptr2) {
            if (ptr1 != arr1) {
                printf(",");
            }
            printf("%d", *ptr1);
            ptr1++;
            ptr2++;
        } else if (*ptr1 < *ptr2) {
            ptr1++;
        } else {
            ptr2++;
        }
```

```
    }

    printf("}\n");
}

int main() {
    int arr1[] = {1, 2, 3, 4, 5, 6};
    int arr2[] = {5, 6, 7, 8};

    int size1 = sizeof(arr1) / sizeof(arr1[0]);
    int size2 = sizeof(arr2) / sizeof(arr2[0]);

    findOverlap(arr1, size1, arr2, size2);

    return 0;
}
```

10. Create a function that rotates an array to the right by `k` elements using pointers.

WTD: Reverse the whole array, then reverse the first k elements, and finally reverse the rest. (e.g.: I/P: int arr[] = {1,2,3,4,5}, k=2 ,O/P: {4,5,1,2,3})

```
#include <stdio.h>

// Function to reverse an array from 'start' to 'end'
void reverse(int arr[], int start, int end) {
    while (start < end) {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

// Function to rotate an array to the right by 'k' elements
void rotateRight(int arr[], int size, int k) {
```

```
    k = k % size; // Ensure 'k' is within the range of array size

    if (k == 0) {
        return; // No need to rotate
    }

    reverse(arr, 0, size - 1);       // Reverse the entire array
    reverse(arr, 0, k - 1);          // Reverse the first 'k' elements
    reverse(arr, k, size - 1);       // Reverse the remaining elements
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    int k = 2; // Number of elements to rotate

    rotateRight(arr, size, k);

    printf("Rotated Array: {");
    for (int i = 0; i < size; i++) {
        printf("%d", arr[i]);
        if (i < size - 1) {
            printf(", ");
        }
    }
    printf("}\n");

    return 0;
}
```

11. Implement a function that merges two sorted arrays into a third array using pointers.

WTD: Use three pointers to traverse and compare the two input arrays and insert the smaller element into the third array.
(e.g.: I/P: int arr1[] = {1,3,5}, arr2[] = {2,4,6} ,O/P: {1,2,3,4,5,6})

```c
#include <stdio.h>

void mergeSortedArrays(int arr1[], int size1, int arr2[], int size2, int
merged[]) {
    int *ptr1 = arr1;
    int *ptr2 = arr2;
    int *ptrMerged = merged;

    while (size1 > 0 && size2 > 0) {
        if (*ptr1 < *ptr2) {
            *ptrMerged = *ptr1;
            ptr1++;
            size1--;
        } else {
            *ptrMerged = *ptr2;
            ptr2++;
            size2--;
        }
        ptrMerged++;
    }

    // If elements are left in arr1, copy them to merged
    while (size1 > 0) {
        *ptrMerged = *ptr1;
        ptr1++;
        ptrMerged++;
        size1--;
    }

    // If elements are left in arr2, copy them to merged
    while (size2 > 0) {
        *ptrMerged = *ptr2;
        ptr2++;
        ptrMerged++;
        size2--;
    }
}

int main() {
    int arr1[] = {1, 3, 5};
```

```
    int arr2[] = {2, 4, 6};
    int size1 = sizeof(arr1) / sizeof(arr1[0]);
    int size2 = sizeof(arr2) / sizeof(arr2[0]);
    int merged[size1 + size2];

    mergeSortedArrays(arr1, size1, arr2, size2, merged);

    printf("Merged Array: ");
    for (int i = 0; i < size1 + size2; i++) {
        printf("%d ", merged[i]);
    }

    return 0;
}
```

12. Design a program that checks if a string is a prefix of another string using pointers.

WTD: Traverse both strings using two pointers. If all characters of the shorter string match the beginning of the longer string, return True.
(e.g.: I/P: char *str1 = "Hello", *str2 = "Hel" ,O/P: True)

```
#include <stdio.h>
#include <stdbool.h>

bool isPrefix(const char *str1, const char *str2) {
    while (*str1 != '\0' && *str2 != '\0') {
        if (*str1 != *str2) {
            return false;  // Characters don't match, not a prefix
        }
        str1++;
        str2++;
    }

    // If str2 is also null-terminated, it means str2 is a prefix of str1
    return (*str2 == '\0');
```

```
}

int main() {
    const char *str1 = "Hello";
    const char *str2 = "Hel";

    if (isPrefix(str1, str2)) {
        printf("%s is a prefix of %s\n", str2, str1);
    } else {
        printf("%s is not a prefix of %s\n", str2, str1);
    }

    return 0;
}
```

13. Write a function that converts a string to lowercase using pointers.

WTD: Traverse the string. For each character, if it's uppercase, add 32 to convert it to lowercase.
(e.g.: I/P: "HELLO" ,O/P: "hello")

```
#include <stdio.h>

void toLowerCase(char *str) {
    while (*str != '\0') {
        if (*str >= 'A' && *str <= 'Z') {
            *str = *str + ('a' - 'A');
        }
        str++;
    }
}

int main() {
    char str[] = "HELLO";

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

```
    toLowerCase(str);
    printf("Lowercase string: %s\n", str);

    return 0;
}
```

14. Implement a program that finds the first non-repeated character in a string using pointers.

WTD: Use a fixed-size array to count occurrences. Traverse the string twice: first to count and then to find the non-repeated character.
(e.g.: I/P: "swiss" ,O/P: 'w')

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

// Function to find the first non-repeated character in a string
char find_first_non_repeating_character(char *str) {
  // Declare a fixed-size array to count the occurrences of each character
  int char_count[256] = {0};

  // Declare two pointers to the beginning and end of the string
  char *p1 = str;
  char *p2 = str + strlen(str);

  // Iterate over the string
  while (p1 < p2) {
    // Increment the count of the current character
    char_count[*p1]++;
    p1++;
  }

  // Declare a pointer to the first non-repeated character
  char *first_non_repeated_character = str;

  // Iterate over the string again
  p1 = str;
  while (p1 < p2) {
```

```
    // If the count of the current character is 1, then it is non-repeated
    if (char_count[*p1] == 1) {
      // Update the pointer to the first non-repeated character
      first_non_repeated_character = p1;
      break;
    }
    p1++;
  }

  // Return the first non-repeated character
  return *first_non_repeated_character;
}

// Driver program
int main() {
  // Declare the string
  char str[] = "swiss";

  // Find the first non-repeated character in the string
  char first_non_repeated_character =
find_first_non_repeating_character(str);

  // Print the result
  printf("The first non-repeated character in the string is '%c'\n",
first_non_repeated_character);

  return 0;
}
```

15. Design a function that uses pointers to find the intersection of two arrays.

WTD: If the arrays are sorted, use two pointers to traverse and find common elements. If not, use a hash table for one array and search for elements of the other.
(e.g.: I/P: int arr1[] = {1,2,3,4}, arr2[] = {3,4,5,6} ,O/P: {3,4})

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

```c
void findIntersection(int arr1[], int size1, int arr2[], int size2) {
    int i = 0, j = 0;

    printf("Intersection: ");

    while (i < size1 && j < size2) {
        if (arr1[i] < arr2[j]) {
            i++;
        } else if (arr1[i] > arr2[j]) {
            j++;
        } else {
            // When elements match, print and increment both pointers
            printf("%d ", arr1[i]);
            i++;
            j++;
        }
    }

    printf("\n");
}

int main() {
    int arr1[] = {1, 2, 3, 4};
    int size1 = sizeof(arr1) / sizeof(arr1[0]);

    int arr2[] = {3, 4, 5, 6};
    int size2 = sizeof(arr2) / sizeof(arr2[0]);

    findIntersection(arr1, size1, arr2, size2);

    return 0;
}
```

16. Create a program that checks if a string is a palindrome using pointers.

WTD: Use two pointers, one at the start and the other at the end. Traverse inward, comparing characters.
(e.g.: I/P: "radar" ,O/P: True)

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

bool isPalindrome(const char *str) {
    int length = strlen(str);

    const char *start = str;
    const char *end = str + length - 1;

    while (start < end) {
        if (*start != *end) {
            return false; // Characters don't match
        }
        start++;
        end--;
    }

    return true; // All characters matched
}

int main() {
    const char *input = "radar";

    if (isPalindrome(input)) {
        printf("'%s' is a palindrome.\n", input);
    } else {
        printf("'%s' is not a palindrome.\n", input);
    }

    return 0;
}
```

17. Write a function to calculate the dot product of two vectors using pointers.

WTD: Traverse both vectors, multiplying corresponding elements and summing the results.
(e.g.: I/P: int vec1[] = {1,2,3}, vec2[] = {4,5,6} ,O/P: 32)

```c
#include <stdio.h>

int calculateDotProduct(const int *vec1, const int *vec2, int length) {
    int dotProduct = 0;

    for (int i = 0; i < length; i++) {
        dotProduct += vec1[i] * vec2[i];
    }

    return dotProduct;
}

int main() {
    int vec1[] = {1, 2, 3};
    int vec2[] = {4, 5, 6};
    int length = sizeof(vec1) / sizeof(vec1[0]);

    int result = calculateDotProduct(vec1, vec2, length);

    printf("Dot Product: %d\n", result);

    return 0;
}
```

18. Design a program that finds the length of a linked list using double pointers.

WTD: Use a double pointer (or two pointers) technique. Move one pointer twice as fast as the other. When the faster one reaches the end, the slower one will be halfway.
(e.g.: I/P:1->2->3->4->5 ,O/P:5)

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

// Define the structure of a node in the linked list
struct Node {
    int data;
    struct Node* next;
};

// Function to find the length of the linked list
int findLinkedListLength(struct Node** head) {
    int length = 0;
    struct Node* current = *head;  // Initialize a current pointer to the head of the list

    while (current != NULL) {
        length++;
        current = current->next;  // Move the current pointer to the next node
    }

    return length;
}

// Function to insert a new node at the end of the linked list
void insertNodeAtEnd(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;

    if (*head == NULL) {
        // If the list is empty, make the new node the head
        *head = newNode;
    } else {
        // Otherwise, traverse to the end and add the new node there
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
```

```
    }
}

// Function to display the linked list
void displayLinkedList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;

    // Insert elements into the linked list
    insertNodeAtEnd(&head, 1);
    insertNodeAtEnd(&head, 2);
    insertNodeAtEnd(&head, 3);
    insertNodeAtEnd(&head, 4);
    insertNodeAtEnd(&head, 5);

    // Display the linked list
    printf("Linked List: ");
    displayLinkedList(head);

    // Find the length of the linked list
    int length = findLinkedListLength(&head);
    printf("Length of the Linked List: %d\n", length);

    return 0;
}
```

19. Implement a function that finds the common elements in three sorted arrays using pointers.

WTD: Use three pointers. Move the pointer pointing to the smallest value until all three pointers point to the same value or the end is reached.
(e.g.: I/P: int arr1[] = {1,5,10,20,40,80}, arr2[] = {6,7,20,80,100}, arr3[] = {3,4,15,20,30,70,80,120} ,O/P:{20,80})

```c
#include <stdio.h>

// Function to find common elements in three sorted arrays
void findCommonElements(int arr1[], int arr2[], int arr3[], int n1, int
n2, int n3) {
    int i = 0, j = 0, k = 0;

    printf("Common Elements: ");

    while (i < n1 && j < n2 && k < n3) {
        if (arr1[i] == arr2[j] && arr2[j] == arr3[k]) {
            printf("%d ", arr1[i]);
            i++;
            j++;
            k++;
        } else if (arr1[i] < arr2[j]) {
            i++;
        } else if (arr2[j] < arr3[k]) {
            j++;
        } else {
            k++;
        }
    }

    printf("\n");
}

int main() {
    int arr1[] = {1, 5, 10, 20, 40, 80};
    int arr2[] = {6, 7, 20, 80, 100};
```

```
    int arr3[] = {3, 4, 15, 20, 30, 70, 80, 120};

    int n1 = sizeof(arr1) / sizeof(arr1[0]);
    int n2 = sizeof(arr2) / sizeof(arr2[0]);
    int n3 = sizeof(arr3) / sizeof(arr3[0]);

    findCommonElements(arr1, arr2, arr3, n1, n2, n3);

    return 0;
}
```

20. Create a program that flattens a 2D array into a 1D array using pointers.

WTD: Use nested loops (or equivalent pointer arithmetic) to traverse the 2D array and copy each element to the 1D array.
(I/P: int arr[][] = {{1,2},{3,4}} ,O/P:{1,2,3,4})

```
#include <stdio.h>

// Function to flatten a 2D array into a 1D array
void flattenArray(int arr[][2], int m, int n, int flatArr[]) {
    int* flatPtr = flatArr; // Pointer to the flat array

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            *flatPtr = arr[i][j]; // Copy the element to the flat array
            flatPtr++; // Move the flat array pointer to the next position
        }
    }
}

int main() {
    int arr[2][2] = {{1, 2}, {3, 4}};
    int m = 2; // Number of rows
```

```
    int n = 2; // Number of columns
    int flatArr[m * n]; // 1D array to store flattened elements

    // Flatten the 2D array into the 1D array
    flattenArray(arr, m, n, flatArr);

    printf("Flattened Array: {");
    for (int i = 0; i < m * n; i++) {
        printf("%d", flatArr[i]);
        if (i < m * n - 1) {
            printf(", ");
        }
    }
    printf("}\n");

    return 0;
}
```

21. Memory Leak Detector. Implement a custom memory allocator and deallocator. Track allocations and deallocations to detect memory leaks using pointers.

WTD: Implement custom allocation and deallocation functions. Maintain a list of allocated blocks. On deallocation, remove from the list. At the end, non-deallocated blocks are leaks. (e.g.: I/P: Allocate 5 blocks, deallocate 4 blocks ; O/P: 1 block not deallocated)

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

// Structure to represent a memory block
typedef struct {
    size_t size; // Size of the allocated memory block
    void* data;  // Pointer to the allocated memory
} MemoryBlock;

MemoryBlock* allocatedBlocks[100]; // Array to store allocated memory
blocks
int numAllocatedBlocks = 0;        // Number of allocated memory blocks
```

```
// Custom memory allocation function
void* customMalloc(size_t size) {
    void* memory = malloc(size);
    if (memory == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
        exit(1);
    }

    // Store information about the allocated block
    MemoryBlock* block = (MemoryBlock*)malloc(sizeof(MemoryBlock));
    block->size = size;
    block->data = memory;

    allocatedBlocks[numAllocatedBlocks++] = block;

    return memory;
}

// Custom memory deallocation function
void customFree(void* ptr) {
    // Find the allocated block corresponding to the pointer
    for (int i = 0; i < numAllocatedBlocks; i++) {
        if (allocatedBlocks[i]->data == ptr) {
            free(ptr);
            free(allocatedBlocks[i]);
            // Remove the block from the list
            for (int j = i; j < numAllocatedBlocks - 1; j++) {
                allocatedBlocks[j] = allocatedBlocks[j + 1];
            }
            numAllocatedBlocks--;
            return;
        }
    }

    fprintf(stderr, "Attempt to free unallocated memory: %p\n", ptr);
}

// Custom memory leak detection function
void detectMemoryLeaks() {
```

```
    if (numAllocatedBlocks > 0) {
        fprintf(stderr, "Memory leaks detected:\n");
        for (int i = 0; i < numAllocatedBlocks; i++) {
            fprintf(stderr, "Block at address %p (size: %zu) was not
deallocated.\n",
                    allocatedBlocks[i]->data, allocatedBlocks[i]->size);
        }
    } else {
        printf("No memory leaks detected.\n");
    }
}

int main() {
    // Allocate 5 memory blocks
    int* arr1 = (int*)customMalloc(5 * sizeof(int));
    int* arr2 = (int*)customMalloc(10 * sizeof(int));
    int* arr3 = (int*)customMalloc(20 * sizeof(int));
    int* arr4 = (int*)customMalloc(30 * sizeof(int));
    int* arr5 = (int*)customMalloc(40 * sizeof(int));

    // Deallocate 4 memory blocks
    customFree(arr1);
    customFree(arr2);
    customFree(arr3);
    customFree(arr4);

    // Detect memory leaks at the end
    detectMemoryLeaks();

    return 0;
}
```

22. Create a structure representing a point in 2D space. Implement a function that moves the point using pointer arithmetic.

WTD: Access the x and y coordinates of the point using pointers and modify them based on the move values.

(e.g.: I/P: Point (2,3), Move by (1,-1) ; O/P: New Point (3,2))

```c
#include <stdio.h>

// Define a structure representing a 2D point
typedef struct {
    int x;
    int y;
} Point;

// Function to move a point by a specified amount using pointer arithmetic
void movePoint(Point* p, int deltaX, int deltaY) {
    // Check if the pointer is valid
    if (p != NULL) {
        // Update the x and y coordinates
        p->x += deltaX;
        p->y += deltaY;
    }
}

int main() {
    // Create a point and initialize its coordinates
    Point myPoint;
    myPoint.x = 2;
    myPoint.y = 3;

    // Print the initial coordinates
    printf("Initial Point: (%d, %d)\n", myPoint.x, myPoint.y);

    // Move the point by (1, -1)
    movePoint(&myPoint, 1, -1);

    // Print the new coordinates after the move
    printf("New Point: (%d, %d)\n", myPoint.x, myPoint.y);

    return 0;
}
```

23. Implement a custom strtok() function using pointers that splits a string based on a delimiter.

WTD: Use pointers to traverse the string. When the delimiter is found, replace it with '\0' and return the start of the token.
(e.g.: I/P: String = "embedded,systems,linux", Delimiter = "," ; O/P: "embedded", "systems", "linux")

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>  // Include the string.h header for strchr

// Custom strtok function
char* customStrtok(char* str, const char* delimiter) {
    static char* nextToken = NULL;  // Pointer to the next token
    if (str != NULL)
        nextToken = str;  // Initialize with the input string

    if (nextToken == NULL || *nextToken == '\0')
        return NULL;  // No more tokens to find

    // Find the start of the token
    char* tokenStart = nextToken;

    // Find the end of the token (delimiter or null terminator)
    while (*nextToken != '\0' && !strchr(delimiter, *nextToken))
        nextToken++;

    if (*nextToken != '\0') {
        *nextToken = '\0';  // Replace delimiter with null terminator
        nextToken++;       // Move to the next character after the
delimiter
    }

    return tokenStart;
}

int main() {
    char input[] = "embedded,systems,linux";
    const char delimiter[] = ",";
```

```
    // Use customStrtok to tokenize the input string
    char* token = customStrtok(input, delimiter);

    while (token != NULL) {
        printf("%s\n", token);
        token = customStrtok(NULL, delimiter);
    }

    return 0;
}
```

24. Design a function similar to memcpy() that copies memory areas using pointers.

WTD: Use pointers to traverse both the source and destination and copy each byte.
(e.g.: I/P: Source = "hello", Destination (empty buffer), Length = 5 ; O/P: Destination = "hello")

```
#include <stdio.h>

// Custom memcpy function
void* customMemcpy(void* dest, const void* src, size_t n) {
    char* charDest = (char*)dest;
    const char* charSrc = (const char*)src;

    // Copy each byte from source to destination
    for (size_t i = 0; i < n; i++) {
        charDest[i] = charSrc[i];
    }

    return dest;
}

int main() {
    // Source string
    const char* source = "hello";
```

```
    // Destination buffer
    char destination[10]; // Make sure it's large enough to hold the data

    // Copy the data from source to destination using customMemcpy
    customMemcpy(destination, source, 5); // Copy 5 characters

    // Null-terminate the destination string (optional)
    destination[5] = '\0';

    // Print the result
    printf("Destination: %s\n", destination); // Should print "hello"

    return 0;
}
```

25. Implement a function that detects if a pointer has gone out of bounds of an array.

WTD: Compare the pointer with the address of the first and past-the-last elements of the array. (e.g.: I/P: Array = {1,2,3,4,5}, Pointer pointing after last element ; O/P: Out of bounds)

```
#include <stdio.h>
#include <stdbool.h>

// Custom function to check if a pointer is out of bounds of an array
bool isPointerOutOfBounds(const int* arr, const int* ptr) {
    const int* firstElement = &arr[0];
    const int* pastTheLastElement = &arr[sizeof(arr) / sizeof(arr[0])];

    return (ptr < firstElement || ptr >= pastTheLastElement);
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int* pointer1 = &arr[2]; // Points to the third element (3)
    int* pointer2 = &arr[5]; // Points after the last element
```

```
    if (isPointerOutOfBounds(arr, pointer1)) {
        printf("Pointer1 is out of bounds.\n");
    } else {
        printf("Pointer1 is within bounds.\n");
    }

    if (isPointerOutOfBounds(arr, pointer2)) {
        printf("Pointer2 is out of bounds.\n");
    } else {
        printf("Pointer2 is within bounds.\n");
    }

    return 0;
}
```

.....Happy Learning .....