Intermediate to Advanced Pointer Problems in C

Problem 1: Pointer Arithmetic for Array Traversal

Problem:

Write a function that finds the sum of elements in an integer array using pointer arithmetic. Do not use array indexing. You are only allowed to use pointer dereferencing and pointer arithmetic.

```
int sum(int *arr, int n);
```

Problem 2: Implementing `strdup` (String Duplicate)

Problem:

Write a function `my_strdup` that duplicates a given string by allocating memory and copying the string into the newly allocated memory. The function should return the pointer to the duplicated string.

```
char *my_strdup(const char *str);
```

- The function should handle `NULL` input by returning `NULL`. - The function should use `malloc` to allocate memory and `memcpy` (or a similar method) to copy the string.

Problem 3: Pointer to Pointer Swap

Problem:

Write a function that swaps the values of two integers using a pointer to pointer approach. You are allowed to pass the addresses of two integer variables using pointer to pointer.

```
void swap(int **x, int **y);
```

Test the function by creating two integer variables and printing their values before and after the swap.

Problem 4: Dynamic Memory Allocation for 2D Array

Problem:

Write a program that dynamically allocates memory for a 2D array of integers. The program should also populate the array with values, and then free the allocated memory.

```
int **create2DArray(int rows, int cols);
void fill2DArray(int **arr, int rows, int cols);
void free2DArray(int **arr, int rows);
```

- `create2DArray` should dynamically allocate memory for the array. - `fill2DArray` should fill the array with values (for example, set `arr[i][j] = i * j`). - `free2DArray` should free the allocated memory.

Problem 5: Implementing `memcpy`

Problem:

Write your own implementation of the `memcpy` function that copies `n` bytes from a source memory location to a destination memory location.

```
void *my_memcpy(void *dest, const void *src, size_t n);
```

- The function should return a pointer to the destination. - Do not use the built-in `memcpy` function. - The function should handle cases where `src` and `dest` may overlap.

Problem 6: Function Pointer to Implement Basic Calculator

Problem:

Create a simple calculator program that uses function pointers. The program should perform addition, subtraction, multiplication, and division using function pointers.

```
int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }
int multiply(int a, int b) { return a * b; }
int divide(int a, int b) { return a / b; }
int (*operation[])(int, int) = {add, subtract, multiply, divide};
```

Test the program by performing each operation on two integers.

Problem 7: Custom `calloc` Implementation

Problem:

Write your own implementation of the `calloc` function. The function should allocate memory for an array of elements and initialize all the bytes to zero.

```
void *my_calloc(size_t num_elements, size_t element_size);
```

- The function should return a pointer to the allocated memory. - If either `num_elements` or `element_size` is `0`, return `NULL`.

Problem 8: Find the Maximum Element in an Array using Pointer Arithmetic

Problem:

Write a function that finds the maximum element in an integer array using pointer arithmetic. Do not use array indexing; only pointer dereferencing and pointer arithmetic should be used.

```
int findMax(int *arr, int n);
```

- The function should return the value of the maximum element in the array.

Problem 9: String Concatenation Using Pointers

Problem:

Write a function that concatenates two strings using pointer manipulation. Do not use any built-in string functions like `strcat`.

```
void my_strcat(char *dest, const char *src);
```

- The function should append the `src` string to the `dest` string. - Make sure to handle null-terminated strings properly.

Problem 10: Detecting Memory Leaks

Problem:

Write a program that simulates a memory leak by allocating memory but failing to free it. Implement a simple function that detects if memory was allocated but not freed (hint: this will be a simplified detection).

```
void memoryLeak(int size) { // Allocate memory and forget to free it }
```

```
int detectLeak() { // Return 1 if leak is detected, otherwise 0 }
```

- The function `detectLeak` will not actually detect real memory leaks but should simulate checking if memory was allocated without freeing it.

Problem 11: Pointer to Function to Execute Custom Operations

Problem:

Write a program that uses pointers to functions to perform different operations on integers. Your program should define a set of operations (e.g., increment, double, square) and execute them based on user input.

```
void increment(int *x) { (*x)++; }

void doubleValue(int *x) { (*x) *= 2; }

void square(int *x) { (*x) = (*x) * (*x); }


int main() { int value = 5; void (*operations[])(int *) = {increment, doubleValue, sq
```

- The program should execute each operation based on user choice and print the result.

Problem 12: Implementing `memcmp`

Problem:

Write your own implementation of the `memcmp` function, which compares two blocks of memory byte by byte.

```
int my_memcmp(const void *str1, const void *str2, size_t n);
```

- The function should return: - `0` if the memory blocks are equal. - A negative value if `str1` is less than `str2`. - A positive value if `str1` is greater than `str2`.

Problem 13: Reversing an Array using Pointer Arithmetic

Problem:

Write a function that reverses an array in place using only pointer arithmetic. Do not use array indices.

```
void reverseArray(int *arr, int n);
```

- The function should reverse the elements of the array `arr` of size `n`.

Problem 14: Circular Buffer Implementation Using Pointers

Implement a circular buffer using pointers. The buffer should support the following operations: 1. Adding an element to the buffer (`enqueue`). 2. Removing an element from the buffer (`dequeue`). 3. Checking if the buffer is full or empty.

```
typedef struct { int *buffer; int head; int tail; int maxSize; } CircularBuffer;
```

Write the following functions:

```
void initBuffer(CircularBuffer *cb, int size);

void enqueue(CircularBuffer *cb, int value);

int dequeue(CircularBuffer *cb);

int isFull(CircularBuffer *cb);

int isEmpty(CircularBuffer *cb);
```

Use dynamic memory allocation to manage the buffer.

Problem 15: Implementing Linked List Traversal Using Pointers

Problem:

Write a program to implement traversal of a singly linked list using pointers. Each node should store an integer value.

```
typedef struct Node { int data; struct Node *next; } Node;
```

Functions to implement:

```
Node *createNode(int value);

void traverseList(Node *head);

void freeList(Node *head);
```

Ensure proper memory management by freeing all nodes at the end of the program.

Problem 16: Pointer-Based Matrix Multiplication

Problem:

Write a program to multiply two matrices using pointers. Use dynamic memory allocation for the matrices.

```
void matrixMultiply(int **mat1, int rows1, int cols1, int **mat2, int rows2, int cols
```

- Validate that the number of columns in `mat1` matches the number of rows in `mat2`.
- Allocate memory for the `result` matrix dynamically.

Problem 17: Detecting Dangling Pointers

Problem:

Write a program to simulate and detect dangling pointers. Create a function that allocates memory, deallocates it, and tries to access the deallocated memory.

```
void simulateDanglingPointer();
```

- The program should demonstrate proper use of `NULL` assignment after deallocation.

Problem 18: Implementing `qsort` Using Function Pointers

Problem:

Implement a simplified version of the `qsort` function using function pointers. The function should sort an array of integers using a comparator function.

```
void my_qsort(int *arr, int n, int (*comparator)(int, int));
```

- Example comparator:

```
int ascending(int a, int b) { return a - b; }

int descending(int a, int b) { return b - a; }
```

Allow the user to choose the sorting order.

Problem 19: Safe Memory Allocation Wrapper

Problem:

Write a wrapper function for `malloc` that ensures safe memory allocation. If the allocation fails, the function should print an error message and terminate the program.

```
void *safeMalloc(size_t size);
```

- Test this wrapper function by allocating memory for an array and initializing it with values.

Problem 20: Pointer-Based Merge Sort

Problem:

Implement the merge sort algorithm for an array of integers using pointers.

```
void mergeSort(int *arr, int left, int right);
void merge(int *arr, int left, int mid, int right);
```

- Use pointer arithmetic for accessing array elements during merging.

Problem 21: Pointer-Based String Reverse

Problem:

Write a function that reverses a string in place using pointer manipulation.

```
void reverseString(char *str);
```

- The function should handle null-terminated strings and ensure proper memory safety.

Problem 22: Multi-Level Pointer Manipulation

Problem:

Write a program to demonstrate multi-level pointers by dynamically creating a variable and modifying its value using a triple pointer.

```
void manipulateTriplePointer(int ***ptr);
```

- Use this program to show the value at the original pointer being changed indirectly.

Problem 23: Custom Dynamic Memory Manager

Problem:

Write a custom dynamic memory manager that manages a fixed-size block of memory. Implement the following functions:

```
void *allocateMemory(size_t size);
void freeMemory(void *ptr);
```

- Use a simple linked list to track allocated and free memory blocks.

Problem 24: Memory Copy with Pointer Overlap Handling

Problem:

Extend the custom `memcpy` implementation to handle cases where the source and destination memory regions overlap.

```
void *my_memcpy_overlap(void *dest, const void *src, size_t n);
```

- Test the function by creating overlapping memory regions and copying data.