Programming in C

Chapter 5: Pointers — Exam-Oriented Programming Problems with Solutions (Set 2)

This document contains a second set of 15 exam and practical/viva programming problems based on the Pointers chapter, each with a complete C program, a line-by-line explanation of the pointer concept used, and a sample output. This set focuses on matrices using pointers, advanced array and string problems, pointer-to-pointer with dynamic 2D allocation, realloc(), a function returning a pointer, and a pointer-based stack.

Problem 1: Write a C program to add two matrices using pointers.

Program:

```
#include <stdio.h>
#define R 2
#define C 2
void addMatrix(int *a, int *b, int *result, int rows, int cols) {
    for (int i = 0; i < rows * cols; i++)
        *(result + i) = *(a + i) + *(b + i);
}
int main() {
    int a[R][C] = {{1, 2}, {3, 4}};
    int b[R][C] = {{5, 6}, {7, 8}};
    int result[R][C];
    addMatrix(&a[0][0], &b[0][0], &result[0][0], R, C);
    printf("Sum matrix:\n");
    for (int i = 0; i < R; i++) {
        for (int j = 0; j < C; j++)
            printf("%d ", result[i][j]);
        printf("\n");
    }
    return 0;
}
```

Explanation:

A two-dimensional array is stored in memory as one contiguous, row-major block of elements. Passing &a[0][0] gives the address of the very first element, so the whole matrix can be treated as a flat one-dimensional array of rows*cols elements and processed with simple pointer arithmetic, *(a + i), instead of double indexing.

Sample Output:

```
Sum matrix:
6 8
10 12
```

Problem 2: Write a C program to multiply two matrices using pointers.

Program:

```
#include <stdio.h>
#define N 2
void multiplyMatrix(int *a, int *b, int *result, int n) {
```

```
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            int sum = 0;
            for (int k = 0; k < n; k++)
                sum += *(a + i * n + k) * *(b + k * n + j);
            *(result + i * n + j) = sum;
        }
    }
}
int main() {
    int a[N][N] = {{1, 2}, {3, 4}};
    int b[N][N] = {{5, 6}, {7, 8}};
    int result[N][N];
    multiplyMatrix(&a[0][0], &b[0][0], &result[0][0], N);
    printf("Product matrix:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf("%d ", result[i][j]);
        printf("\n");
    }
    return 0;
}
```

Explanation:

For a matrix flattened into a 1D array, the element at row i, column j is located at offset i*n + j. The expressions *(a + i*n + k) and *(b + k*n + j) use this formula to fetch the correct elements for the standard row-by-column multiplication, with the running total stored back through *(result + i*n + j).

Sample Output:

```
Product matrix:
19 22
43 50
```

Problem 3: Write a C program to find the transpose of a matrix using pointers.

Program:

```
#include <stdio.h>
#define R 2
#define C 3
void transpose(int *a, int *t, int rows, int cols) {
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            *(t + j * rows + i) = *(a + i * cols + j);
}
int main() {
    int a[R][C] = {{1, 2, 3}, {4, 5, 6}};
    int t[C][R];
    transpose(&a[0][0], &t[0][0], R, C);
    printf("Transposed matrix:\n");
    for (int i = 0; i < C; i++) {
        for (int j = 0; j < R; j++)
            printf("%d ", t[i][j]);
        printf("\n");
    }
```

```
    return 0;
}
```

Explanation:

Transposing swaps rows and columns. The pointer expression *(t + j*rows + i) = *(a + i*cols + j) copies each element from its original position in the row-major source matrix to its mirrored row/column position in the destination matrix, again using only pointer offsets on the flattened arrays.

Sample Output:

```
Transposed matrix:
1 4
2 5
3 6
```

Problem 4: Write a C program to find the second largest element in an array using pointers.

Program:

```
#include <stdio.h>
int secondLargest(int *arr, int n) {
    int first = *arr, second = -1;
    for (int i = 1; i < n; i++) {
        int val = *(arr + i);
        if (val > first) {
            second = first;
            first = val;
        } else if (val > second && val != first) {
            second = val;
        }
    }
    return second;
}
int main() {
    int arr[] = {12, 35, 1, 10, 34, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Second largest = %d", secondLargest(arr, n));
    return 0;
}
```

Explanation:

The function keeps track of the largest (first) and second largest (second) values seen so far, purely by dereferencing the array through the pointer arr (*(arr + i)) as it scans once through all the elements — no sorting is required.

Sample Output:

```
Second largest = 34
```

Problem 5: Write a C program to count the number of even and odd elements in an array using pointers.

Program:

```c
#include <stdio.h>
void countEvenOdd(int *arr, int n, int *even, int *odd) {
    *even = *odd = 0;
    for (int i = 0; i < n; i++) {
        if (*(arr + i) % 2 == 0)
            (*even)++;
        else
            (*odd)++;
    }
}
int main() {
    int arr[] = {10, 15, 22, 33, 40, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    int even, odd;
    countEvenOdd(arr, n, &even, &odd);
    printf("Even = %d\n", even);
    printf("Odd = %d", odd);
    return 0;
}
```

Explanation:

The counts even and odd are passed to the function as pointers (&even, &odd) so that a single function call can update two separate results in the caller. Inside the loop, (*even)++ and (*odd)++ increment the values pointed to, not the pointers themselves — note the parentheses, which are essential because ++ binds tighter than *.

Sample Output:

```
Even = 3
Odd = 3
```

Problem 6: Write a C program to merge two sorted arrays into a single sorted array using pointers.

Program:

```c
#include <stdio.h>
void mergeArrays(int *a, int n, int *b, int m, int *result) {
    int i = 0, j = 0, k = 0;
    while (i < n && j < m) {
        if (*(a + i) <= *(b + j))
            *(result + k++) = *(a + i++);
        else
            *(result + k++) = *(b + j++);
    }
    while (i < n) *(result + k++) = *(a + i++);
    while (j < m) *(result + k++) = *(b + j++);
}
int main() {
    int a[] = {1, 3, 5, 7};
    int b[] = {2, 4, 6, 8, 10};
    int n = sizeof(a) / sizeof(a[0]);
    int m = sizeof(b) / sizeof(b[0]);
    int result[9];
    mergeArrays(a, n, b, m, result);
```

```
    printf("Merged array: ");
    for (int i = 0; i < n + m; i++)
        printf("%d ", result[i]);
    return 0;
}
```

Explanation:

Three pointers-turned-indices (i, j, k) track the current position in array a, array b, and the merged result respectively. At each step, the smaller of *(a + i) and *(b + j) is written into *(result + k), and the corresponding index is advanced — the standard merge step of merge sort, expressed with pointer arithmetic.

Sample Output:

```
Merged array: 1 2 3 4 5 6 7 8 10
```

Problem 7: Write a C program to check whether a string is a palindrome using pointers.

Program:

```
#include <stdio.h>
#include <string.h>
int isPalindrome(char *str) {
    char *start = str;
    char *end = str + strlen(str) - 1;
    while (start < end) {
        if (*start != *end)
            return 0;
        start++;
        end--;
    }
    return 1;
}
int main() {
    char str[] = "madam";
    if (isPalindrome(str))
        printf("\"%s\" is a palindrome", str);
    else
        printf("\"%s\" is not a palindrome", str);
    return 0;
}
```

Explanation:

Two pointers, start and end, move towards each other from opposite ends of the string, comparing the characters they point to at every step. If any pair of characters differs, the string cannot be a palindrome and the function returns immediately; otherwise the pointers eventually meet and the string is confirmed to be a palindrome.

Sample Output:

```
"madam" is a palindrome
```

Problem 8: Write a C program to count the number of words in a sentence using pointers.

Program:

```
#include <stdio.h>
int countWords(char *str) {
    int count = 0;
    int inWord = 0;
    while (*str != '\0') {
        if (*str != ' ' && *str != '\t' && !inWord) {
            inWord = 1;
            count++;
        } else if (*str == ' ' || *str == '\t') {
            inWord = 0;
        }
        str++;
    }
    return count;
}
int main() {
    char sentence[] = "Pointers make C programming powerful";
    printf("Number of words = %d", countWords(sentence));
    return 0;
}
```

Explanation:

The pointer str walks through the sentence one character at a time. A word is counted only at the transition from a space (or the very start) into a non-space character, tracked using the flag inWord, so that consecutive letters within the same word are not counted more than once.

Sample Output:

```
Number of words = 5
```

Problem 9: Write a C program to concatenate two strings using pointers (without using strcat()).

Program:

```
#include <stdio.h>
void stringConcat(char *dest, char *src) {
    while (*dest != '\0')
        dest++;
    while (*src != '\0') {
        *dest = *src;
        dest++;
        src++;
    }
    *dest = '\0';
}
int main() {
    char str1[50] = "Programming in ";
    char str2[] = "C";
    stringConcat(str1, str2);
    printf("Concatenated string: %s", str1);
    return 0;
}
```

Explanation:

The pointer dest is first advanced past the existing characters of str1 until it reaches its terminating '\0'. From that position, characters from str2 are copied one by one through the src pointer, and a fresh null terminator is placed at the end once copying is complete.

Sample Output:

```
Concatenated string: Programming in C
```

Problem 10: Write a C program to find the frequency of a given character in a string using pointers.

Program:

```
#include <stdio.h>
int countChar(char *str, char ch) {
    int count = 0;
    while (*str != '\0') {
        if (*str == ch)
            count++;
        str++;
    }
    return count;
}
int main() {
    char str[] = "programming";
    char ch = 'g';
    printf("'%c' occurs %d times in \"%s\"", ch, countChar(str, ch), str);
    return 0;
}
```

Explanation:

The pointer str is advanced through the string one character at a time, and each character it points to (*str) is compared with the target character ch, incrementing a counter on every match until the terminating null character is reached.

Sample Output:

```
'g' occurs 2 times in "programming"
```

Problem 11: Write a C program to sort an array of strings alphabetically using an array of pointers (by swapping pointers, not the strings themselves).

Program:

```
#include <stdio.h>
#include <string.h>
void sortStrings(char *arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (strcmp(arr[j], arr[j + 1]) > 0) {
                char *temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
```

```
    }
}
int main() {
    char *names[] = {"Rahul", "Amit", "Sneha", "Priya"};
    int n = sizeof(names) / sizeof(names[0]);
    sortStrings(names, n);
    printf("Sorted names:\n");
    for (int i = 0; i < n; i++)
        printf("%s\n", names[i]);
    return 0;
}
```

Explanation:

Because names is an array of character pointers, sorting it only requires swapping the pointers themselves (char *temp = arr[j]; ...) rather than copying the characters of each string. This makes the sort efficient regardless of how long the individual strings are, since only addresses are exchanged.

Sample Output:

```
Sorted names:
Amit
Priya
Rahul
Sneha
```

Problem 12: Write a C program in which a function returns a pointer to the maximum element of an array.

Program:

```
#include <stdio.h>
int *findMax(int *arr, int n) {
    int *maxPtr = arr;
    for (int i = 1; i < n; i++) {
        if (*(arr + i) > *maxPtr)
            maxPtr = arr + i;
    }
    return maxPtr;
}
int main() {
    int arr[] = {45, 67, 12, 89, 33};
    int n = sizeof(arr) / sizeof(arr[0]);
    int *max = findMax(arr, n);
    printf("Maximum element = %d\n", *max);
    printf("Found at index = %ld", max - arr);
    return 0;
}
```

Explanation:

findMax() is declared to return int * rather than int, so it returns the address of the maximum element within the array itself, not just its value. Subtracting the base address of the array from this returned pointer (max - arr) gives the index of the maximum element, a valid use of pointer subtraction between two pointers into the same array.

Sample Output:

```
Maximum element = 89
Found at index = 3
```

Problem 13: Write a C program to dynamically allocate a two-dimensional matrix using pointer to pointer and malloc().

Program:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int rows = 2, cols = 3, i, j;
    int **matrix = (int **) malloc(rows * sizeof(int *));
    for (i = 0; i < rows; i++)
        matrix[i] = (int *) malloc(cols * sizeof(int));
    int value = 1;
    for (i = 0; i < rows; i++)
        for (j = 0; j < cols; j++)
            matrix[i][j] = value++;
    printf("Dynamically allocated matrix:\n");
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++)
            printf("%d ", matrix[i][j]);
        printf("\n");
    }
    for (i = 0; i < rows; i++)
        free(matrix[i]);
    free(matrix);
    return 0;
}
```

Explanation:

matrix is a pointer to pointer (int **). The first malloc() allocates an array of row pointers, and a separate malloc() inside the loop allocates each row individually, so the matrix's dimensions can be decided entirely at run time rather than being fixed at compile time. Each row is freed individually before the row-pointer array itself is freed, to avoid a memory leak.

Sample Output:

```
Dynamically allocated matrix:
1 2 3
4 5 6
```

Problem 14: Write a C program to demonstrate the use of realloc() to resize a dynamically allocated array at run time.

Program:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n = 3, i;
    int *arr = (int *) malloc(n * sizeof(int));
    for (i = 0; i < n; i++)
```

```
        arr[i] = (i + 1) * 10;
    printf("Original array: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    n = 5;
    arr = (int *) realloc(arr, n * sizeof(int));
    if (arr == NULL) {
        printf("\nMemory reallocation failed");
        return 1;
    }
    for (i = 3; i < n; i++)
        arr[i] = (i + 1) * 10;
    printf("\nResized array: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    free(arr);
    return 0;
}
```

Explanation:

The array is first allocated with malloc() to hold 3 integers. Calling realloc(arr, n * sizeof(int)) after changing n to 5 resizes the same block of memory to hold 5 integers, preserving the values already stored in the first 3 positions, after which the 2 newly available slots are filled in.

Sample Output:

```
Original array: 10 20 30
Resized array: 10 20 30 40 50
```

Problem 15: Write a C program to implement a simple array-based stack (push and pop) using pointers.

Program:

```
#include <stdio.h>
#define MAX 5
int stack[MAX];
int *top = stack - 1;   // one before the base: stack is empty
void push(int value) {
    if (top == stack + MAX - 1) {
        printf("Stack overflow\n");
        return;
    }
    top++;
    *top = value;
}
int pop() {
    if (top < stack) {
        printf("Stack underflow\n");
        return -1;
    }
    int value = *top;
    top--;
    return value;
}
int main() {
```

```
    push(10);
    push(20);
    push(30);
    printf("Popped: %d\n", pop());
    printf("Popped: %d\n", pop());
    push(40);
    printf("Popped: %d\n", pop());
    printf("Popped: %d\n", pop());
    return 0;
}
```

Explanation:

top is a pointer that always marks the current top of the stack, starting one position before the base (stack - 1) to represent an empty stack. push() advances top and stores the new value at *top; pop() reads the value at *top and then moves top backward — the entire last-in-first-out behaviour of the stack is implemented purely through pointer movement and dereferencing.

Sample Output:

```
Popped: 30
Popped: 20
Popped: 40
Popped: 10
```