Programming in C

Chapter 5: Pointers — Exam-Oriented Programming Problems with Solutions

This document contains 15 frequently asked 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. The problems cover pointer basics, pointers with functions and arrays, pointers and strings, array of pointers, function pointers, pointer to pointer, and dynamic memory allocation.

Problem 1: Write a C program to swap two numbers using pointers (call by reference).

Program:

```
#include <stdio.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main() {
    int x, y;
    printf("Enter two integers: ");
    scanf("%d %d", &x, &y);
    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;
}
```

Explanation:

The addresses of x and y (&x and &y) are passed to swap(), so the parameters a and b themselves become pointers to x and y. Inside the function, *a and *b dereference these pointers, so modifying *a and *b directly modifies x and y in main() — this is call by reference, made possible only through pointers.

Sample Output:

```
Enter two integers: 5 10
Before swap: x = 5, y = 10
After swap: x = 10, y = 5
```

Problem 2: Write a C program to find the largest of three numbers using pointers.

Program:

```
#include <stdio.h>
int largest(int *a, int *b, int *c) {
    int max = *a;
    if (*b > max) max = *b;
    if (*c > max) max = *c;
    return max;
}
int main() {
    int a, b, c;
    printf("Enter three integers: ");
```

```
    scanf("%d %d %d", &a, &b, &c);
    printf("Largest number is %d", largest(&a, &b, &c));
    return 0;
}
```

Explanation:

The addresses of a, b, and c are passed to largest(), which receives them as pointers and dereferences each one (*a, *b, *c) to compare their values. Only the addresses — not copies of the whole variables — need to be transferred to the function.

Sample Output:

```
Enter three integers: 4 9 7
Largest number is 9
```

Problem 3: Write a C program to find the sum and average of array elements using pointer arithmetic.

Program:

```
#include <stdio.h>
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    int *p = arr;
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += *(p + i);
    float avg = (float) sum / n;
    printf("Sum = %d\n", sum);
    printf("Average = %.2f", avg);
    return 0;
}
```

Explanation:

The array name arr decays to a pointer to its first element, which is assigned to p. The expression *(p + i) uses pointer arithmetic to access the i-th element (equivalent to arr[i] or p[i]), and each addition of i automatically advances the pointer by i x sizeof(int) bytes.

Sample Output:

```
Sum = 150
Average = 30.00
```

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

Program:

```
#include <stdio.h>
void findMinMax(int *arr, int n, int *min, int *max) {
    *min = *max = arr[0];
    for (int i = 1; i < n; i++) {
        if (*(arr + i) < *min) *min = *(arr + i);
        if (*(arr + i) > *max) *max = *(arr + i);
    }
```

```
}
int main() {
    int arr[] = {23, 5, 78, 12, 45};
    int n = sizeof(arr) / sizeof(arr[0]);
    int min, max;
    findMinMax(arr, n, &min, &max);
    printf("Smallest = %d\n", min);
    printf("Largest = %d", max);
    return 0;
}
```

Explanation:

The array is passed as a pointer (arr), while min and max are passed as pointers (&min, &max) so that the function can return two results at once by writing directly into the caller's variables through *min and *max — something an ordinary return value cannot do.

Sample Output:

```
Smallest = 5
Largest = 78
```

Problem 5: Write a C program to reverse the elements of an array using pointers.

Program:

```
#include <stdio.h>
void reverseArray(int *arr, int n) {
    int *start = arr;
    int *end = arr + n - 1;
    while (start < end) {
        int temp = *start;
        *start = *end;
        *end = temp;
        start++;
        end--;
    }
}
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    reverseArray(arr, n);
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Explanation:

Two pointers, start and end, are set to the first and last elements of the array. The values they point to are swapped, and the pointers are moved towards each other (start++ and end--) until they meet, reversing the array in place without using any extra array.

Sample Output:

```
5 4 3 2 1
```

Problem 6: Write a C program to perform a linear search on an array using pointers.

Program:

```
#include <stdio.h>
int linearSearch(int *arr, int n, int key) {
    for (int i = 0; i < n; i++) {
        if (*(arr + i) == key)
            return i;
    }
    return -1;
}
int main() {
    int arr[] = {15, 22, 39, 47, 8};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 39;
    int pos = linearSearch(arr, n, key);
    if (pos != -1)
        printf("Element %d found at index %d", key, pos);
    else
        printf("Element not found");
    return 0;
}
```

Explanation:

The function receives the array as a pointer (arr) and steps through it using pointer arithmetic (*(arr + i)), comparing each element with the search key until a match is found or the end of the array is reached.

Sample Output:

```
Element 39 found at index 2
```

Problem 7: Write a C program to sort an array in ascending order using pointers (bubble sort).

Program:

```
#include <stdio.h>
void bubbleSort(int *arr, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (*(arr + j) > *(arr + j + 1)) {
                int temp = *(arr + j);
                *(arr + j) = *(arr + j + 1);
                *(arr + j + 1) = temp;
            }
        }
    }
}
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Explanation:

The bubble sort logic compares and swaps adjacent elements using pointer expressions *(arr + j) and *(arr + j + 1) instead of array indexing. Because the array is passed as a pointer, the swaps directly modify the original array in main().

Sample Output:

```
11 12 22 25 64
```

Problem 8: Write a C program to find the length of a string using pointers (without using strlen()).

Program:

```
#include <stdio.h>
int stringLength(char *str) {
    int len = 0;
    while (*(str + len) != '\0')
        len++;
    return len;
}
int main() {
    char str[] = "Pointers";
    printf("Length of string = %d", stringLength(str));
    return 0;
}
```

Explanation:

Since a string is simply a character array terminated by the null character '\0', the function walks through the string using a pointer, incrementing a counter until *(str + len) equals '\0', which marks the end of the string.

Sample Output:

```
Length of string = 8
```

Problem 9: Write a C program to reverse a string using pointers (without using a built-in function).

Program:

```
#include <stdio.h>
#include <string.h>
void reverseString(char *str) {
    char *start = str;
    char *end = str + strlen(str) - 1;
    while (start < end) {
        char temp = *start;
        *start = *end;
        *end = temp;
        start++;
        end--;
    }
}
int main() {
    char str[] = "POINTER";
    reverseString(str);
```

```
    printf("Reversed string: %s", str);
    return 0;
}
```

Explanation:

Two character pointers, start and end, are positioned at the beginning and end of the string. Their contents are swapped and the pointers are moved towards the centre, reversing the string in place — the same two-pointer technique used to reverse an array.

Sample Output:

```
Reversed string: RETNIOP
```

Problem 10: Write a C program to copy the contents of one string into another using pointers (without using strcpy()).

Program:

```
#include <stdio.h>
void stringCopy(char *dest, char *src) {
    while (*src != '\0') {
        *dest = *src;
        dest++;
        src++;
    }
    *dest = '\0';
}
int main() {
    char source[] = "Engineering";
    char destination[50];
    stringCopy(destination, source);
    printf("Copied string: %s", destination);
    return 0;
}
```

Explanation:

The function copies characters one at a time from src to dest by dereferencing both pointers, advancing each pointer after every character, and finally appending a null character ('\0') to properly terminate the copied string.

Sample Output:

```
Copied string: Engineering
```

Problem 11: Write a C program to count the number of vowels and consonants in a string using pointers.

Program:

```
#include <stdio.h>
#include <ctype.h>
int main() {
    char str[] = "Programming in C";
    char *p = str;
```

```
    int vowels = 0, consonants = 0;
    while (*p != '\0') {
        char ch = tolower(*p);
        if (ch=='a'||ch=='e'||ch=='i'||ch=='o'||ch=='u')
            vowels++;
        else if (isalpha(ch))
            consonants++;
        p++;
    }
    printf("Vowels = %d\n", vowels);
    printf("Consonants = %d", consonants);
    return 0;
}
```

Explanation:

A pointer p is initialised to the start of the string and moved forward one character at a time (p++). Each character it points to (*p) is checked and classified as a vowel, a consonant, or neither (such as a space), without ever using array-index notation.

Sample Output:

```
Vowels = 4
Consonants = 10
```

Problem 12: Write a C program to store and display a list of names using an array of pointers to strings.

Program:

```
#include <stdio.h>
int main() {
    char *names[] = {"Amit", "Priya", "Rahul", "Sneha"};
    int n = sizeof(names) / sizeof(names[0]);
    for (int i = 0; i < n; i++)
        printf("%d. %s\n", i + 1, names[i]);
    return 0;
}
```

Explanation:

names is an array of character pointers, where each element stores the address of a separate string literal of its own length. This is more memory-efficient than a fixed-width two-dimensional character array when the strings are of different lengths.

Sample Output:

```
1. Amit
2. Priya
3. Rahul
4. Sneha
```

Problem 13: Write a C program to perform basic arithmetic operations using function pointers.

Program:

```c
#include <stdio.h>
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a * b; }
int main() {
    int (*operation[3])(int, int) = {add, sub, mul};
    char *labels[3] = {"Addition", "Subtraction", "Multiplication"};
    int a = 12, b = 4;
    for (int i = 0; i < 3; i++)
        printf("%s: %d\n", labels[i], operation[i](a, b));
    return 0;
}
```

Explanation:

operation is an array of function pointers, each element storing the address of a different function (add, sub, mul). The correct function is invoked indirectly through operation[i](a, b), demonstrating how function pointers allow behaviour to be selected and called at run time, as in a dispatch table.

Sample Output:

```
Addition: 16
Subtraction: 8
Multiplication: 48
```

Problem 14: Write a C program to demonstrate the working of a pointer to a pointer (double pointer).

Program:

```c
#include <stdio.h>
int main() {
    int value = 25;
    int *p = &value;
    int **pp = &p;
    printf("Value (direct)     = %d\n", value);
    printf("Value via *p        = %d\n", *p);
    printf("Value via **pp      = %d\n", **pp);
    return 0;
}
```

Explanation:

p is a pointer that stores the address of value, and pp is a pointer to a pointer that stores the address of p itself. Dereferencing pp once (*pp) gives back p, and dereferencing it twice (**pp) gives back value — this is the double indirection used, for example, when a function needs to modify a pointer's value in the caller.

Sample Output:

```
Value (direct)     = 25
Value via *p        = 25
Value via **pp      = 25
```

Problem 15: Write a C program to dynamically allocate an array using malloc(), and calculate the sum and average of its elements.

Program:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n, i, sum = 0;
    int *arr;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    arr = (int *) malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed");
        return 1;
    }
    printf("Enter %d integers: ", n);
    for (i = 0; i < n; i++)
        scanf("%d", (arr + i));
    for (i = 0; i < n; i++)
        sum += *(arr + i);
    printf("Sum = %d\n", sum);
    printf("Average = %.2f\n", (float) sum / n);
    free(arr);
    return 0;
}
```

Explanation:

Instead of a fixed-size array, malloc(n * sizeof(int)) requests memory for exactly n integers at run time, and arr holds the address of this block. Elements are read and summed using pointer arithmetic (arr + i), and free(arr) releases the memory once it is no longer needed, avoiding a memory leak.

Sample Output:

```
Enter number of elements: 5
Enter 5 integers: 10 20 30 40 50
Sum = 150
Average = 30.00
```