Programming in C

Chapter 5: Pointers — Exam-Oriented Programming Problems with Solutions (Set 3)

This document contains a third 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 pointer arithmetic across data types, void pointers, array-to-pointer decay, array insertion/deletion/deduplication, set operations, word and case manipulation in strings, a queue implemented with pointers, and the important viva distinctions between a pointer to an array vs. an array of pointers, and a pointer to a constant vs. a constant pointer.

Problem 1: Write a C program to demonstrate pointer arithmetic on different data types and show how the address changes when the pointer is incremented.

Program:

```
#include <stdio.h>
int main() {
    int i[3];
    float f[3];
    double d[3];
    char c[3];
    printf("int:    &i[0]=%p  &i[1]=%p  (size = %lu bytes)\n", (void*)&i[0],
(void*)&i[1], sizeof(int));
    printf("float:  &f[0]=%p  &f[1]=%p  (size = %lu bytes)\n", (void*)&f[0],
(void*)&f[1], sizeof(float));
    printf("double: &d[0]=%p  &d[1]=%p  (size = %lu bytes)\n", (void*)&d[0],
(void*)&d[1], sizeof(double));
    printf("char:   &c[0]=%p  &c[1]=%p  (size = %lu bytes)\n", (void*)&c[0],
(void*)&c[1], sizeof(char));
    return 0;
}
```

Explanation:

When a pointer of a given type is incremented, it advances by exactly sizeof(type) bytes, not by one byte. This program prints the addresses of consecutive elements of arrays of different types to show that the gap between &x[0] and &x[1] always equals sizeof(type) — 4 bytes apart for int and float, 8 for double, and 1 for char on a typical system.

Sample Output:

```
int:    &i[0]=0x61fe1c  &i[1]=0x61fe20  (size = 4 bytes)
float:  &f[0]=0x61fe10  &f[1]=0x61fe14  (size = 4 bytes)
double: &d[0]=0x61fe00  &d[1]=0x61fe08  (size = 8 bytes)
char:   &c[0]=0x61fe2f  &c[1]=0x61fe30  (size = 1 byte)
(Exact addresses vary on every run and system; the constant gap matching sizeof(type) is
the key point.)
```

Problem 2: Write a C program to demonstrate the use of a generic (void) pointer to point to variables of different data types.

Program:

```
#include <stdio.h>
int main() {
    int i = 10;
    float f = 5.5;
    char c = 'A';
    void *gp;
    gp = &i;
    printf("Integer value   = %d\n", *(int *) gp);
    gp = &f;
    printf("Float value     = %.2f\n", *(float *) gp);
    gp = &c;
    printf("Character value = %c\n", *(char *) gp);
    return 0;
}
```

Explanation:

The same void pointer gp is reused to hold the address of an int, then a float, then a char. Because the compiler does not know what type a void pointer refers to, each dereference must be explicitly cast to the correct type — (int *) gp, (float *) gp, (char *) gp — before the value can be read.

Sample Output:

```
Integer value   = 10
Float value     = 5.50
Character value = A
```

Problem 3: Write a C program to demonstrate that an array decays into a pointer when passed to a function, by comparing sizeof(array) inside and outside the function.

Program:

```
#include <stdio.h>
void checkSize(int arr[]) {
    printf("Inside function, sizeof(arr) = %lu bytes (pointer size)\n", sizeof(arr));
}
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    printf("Inside main, sizeof(arr) = %lu bytes (whole array)\n", sizeof(arr));
    checkSize(arr);
    return 0;
}
```

Explanation:

In main(), arr refers to the actual array, so sizeof(arr) gives the total size of all 5 integers (20 bytes on a system with a 4-byte int). Once arr is passed to checkSize(), it decays into a pointer to its first element, so inside the function sizeof(arr) instead reports the size of a single pointer (commonly 8 bytes on a 64-bit system) — proof that the function never receives the full array, only its address.

Sample Output:

```
Inside main, sizeof(arr) = 20 bytes (whole array)
Inside function, sizeof(arr) = 8 bytes (pointer size)
```

Problem 4: Write a C program to insert an element at a given position in an array using pointers.

Program:

```
#include <stdio.h>
void insertElement(int *arr, int *n, int pos, int value) {
    for (int i = *n; i > pos; i--)
        *(arr + i) = *(arr + i - 1);
    *(arr + pos) = value;
    (*n)++;
}
int main() {
    int arr[10] = {10, 20, 30, 40};
    int n = 4;
    insertElement(arr, &n, 2, 99);
    printf("Array after insertion: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Explanation:

The elements from the end of the array down to the insertion position are shifted one place to the right using *(arr + i) = *(arr + i - 1), which creates a gap at index pos. The new value is then written into that gap, and n — passed as a pointer (&n) so the updated count is visible in main() — is incremented.

Sample Output:

```
Array after insertion: 10 20 99 30 40
```

Problem 5: Write a C program to delete an element from a given position in an array using pointers.

Program:

```
#include <stdio.h>
void deleteElement(int *arr, int *n, int pos) {
    for (int i = pos; i < *n - 1; i++)
        *(arr + i) = *(arr + i + 1);
    (*n)--;
}
int main() {
    int arr[] = {10, 20, 99, 30, 40};
    int n = 5;
    deleteElement(arr, &n, 2);
    printf("Array after deletion: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Explanation:

Starting from the position to be deleted, each subsequent element is shifted one place to the left using *(arr + i) = *(arr + i + 1), overwriting the deleted value. The element count n, again received as a pointer, is decremented to reflect the array's new logical size.

Sample Output:

```
Array after deletion: 10 20 30 40
```

Problem 6: Write a C program to remove duplicate elements from an array using pointers.

Program:

```
#include <stdio.h>
int removeDuplicates(int *arr, int n) {
    int *end = arr + n;
    int *write = arr + 1;
    for (int *read = arr + 1; read < end; read++) {
        int isDup = 0;
        for (int *p = arr; p < write; p++) {
            if (*p == *read) { isDup = 1; break; }
        }
        if (!isDup) {
            *write = *read;
            write++;
        }
    }
    return write - arr;
}
int main() {
    int arr[] = {10, 20, 10, 30, 20, 40};
    int n = sizeof(arr) / sizeof(arr[0]);
    int newLen = removeDuplicates(arr, n);
    printf("Array after removing duplicates: ");
    for (int i = 0; i < newLen; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Explanation:

Two pointers, read and write, both scan the array. read examines every element in order, while write marks where the next unique element should be stored. Each element under read is checked against everything already written (arr up to write); only new values are copied to *write, and write - arr at the end gives the new, deduplicated length.

Sample Output:

```
Array after removing duplicates: 10 20 30 40
```

Problem 7: Write a C program to find the union and intersection of two arrays using pointers.

Program:

```
#include <stdio.h>
int main() {
    int a[] = {1, 2, 3, 4, 5};
    int b[] = {3, 4, 5, 6, 7};
    int na = sizeof(a) / sizeof(a[0]);
    int nb = sizeof(b) / sizeof(b[0]);
    int *pa = a, *pb = b;
```

```
    printf("Union: ");
    for (int i = 0; i < na; i++)
        printf("%d ", *(pa + i));
    for (int i = 0; i < nb; i++) {
        int found = 0;
        for (int j = 0; j < na; j++)
            if (*(pb + i) == *(pa + j)) { found = 1; break; }
        if (!found)
            printf("%d ", *(pb + i));
    }
    printf("\nIntersection: ");
    for (int i = 0; i < na; i++) {
        for (int j = 0; j < nb; j++) {
            if (*(pa + i) == *(pb + j)) {
                printf("%d ", *(pa + i));
                break;
            }
        }
    }
    return 0;
}
```

Explanation:

Two pointers, pa and pb, walk through arrays a and b respectively. For the union, every element of a is printed first, followed by only the elements of b that do not already appear in a (checked via a nested scan). For the intersection, only the elements of a that also appear somewhere in b are printed.

Sample Output:

```
Union: 1 2 3 4 5 6 7
Intersection: 3 4 5
```

Problem 8: Write a C program to find the sum of the diagonal elements of a square matrix using pointers.

Program:

```
#include <stdio.h>
int main() {
    int n = 3;
    int mat[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int *p = &mat[0][0];
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += *(p + i * n + i);
    printf("Sum of diagonal elements = %d", sum);
    return 0;
}
```

Explanation:

For an n x n matrix flattened into a single block of memory, the diagonal element in row i (which is also column i) sits at offset i*n + i from the start. The pointer p, initialised to &mat[0][0], is used with this formula to add up mat[0][0], mat[1][1], and mat[2][2] without ever using double-index notation.

Sample Output:

```
Sum of diagonal elements = 15
```

Problem 9: Write a C program to reverse the order of words in a sentence using pointers.

Program:

```
#include <stdio.h>
#include <string.h>
void reverseWords(char *str) {
    char temp[100];
    char *words[20];
    int count = 0;
    char *token = strtok(str, " ");
    while (token != NULL) {
        words[count++] = token;
        token = strtok(NULL, " ");
    }
    temp[0] = '\0';
    for (int i = count - 1; i >= 0; i--) {
        strcat(temp, words[i]);
        if (i != 0)
            strcat(temp, " ");
    }
    strcpy(str, temp);
}
int main() {
    char sentence[100] = "Pointers are powerful in C";
    reverseWords(sentence);
    printf("Reversed sentence: %s", sentence);
    return 0;
}
```

Explanation:

strtok() splits the sentence into individual words, and words[] — an array of character pointers — stores the address of each word within the original string. The words are then written into a new buffer starting from the last pointer stored (words[count-1]) down to the first, effectively reversing the order of the words rather than the characters within them.

Sample Output:

```
Reversed sentence: C in powerful are Pointers
```

Problem 10: Write a C program to convert all lowercase letters in a string to uppercase using pointers.

Program:

```
#include <stdio.h>
int main() {
    char str[] = "Programming in C";
    char *p = str;
    while (*p != '\0') {
        if (*p >= 'a' && *p <= 'z')
            *p = *p - 'a' + 'A';
```

```
        p++;
    }
    printf("Uppercase string: %s", str);
    return 0;
}
```

Explanation:

The pointer p moves through the string one character at a time. Whenever *p is a lowercase letter, the expression *p - 'a' + 'A' converts it to the corresponding uppercase letter by using the fixed distance between the lowercase and uppercase letter ranges in the character set, and the result is written back through *p, modifying the string in place.

Sample Output:

```
Uppercase string: PROGRAMMING IN C
```

Problem 11: Write a C program to check whether two strings are anagrams of each other using pointers.

Program:

```
#include <stdio.h>
#include <string.h>
int isAnagram(char *s1, char *s2) {
    int count[256] = {0};
    char *p1 = s1, *p2 = s2;
    if (strlen(s1) != strlen(s2))
        return 0;
    while (*p1 != '\0') {
        count[(int) *p1]++;
        p1++;
    }
    while (*p2 != '\0') {
        count[(int) *p2]--;
        p2++;
    }
    for (int i = 0; i < 256; i++)
        if (count[i] != 0)
            return 0;
    return 1;
}
int main() {
    char s1[] = "listen";
    char s2[] = "silent";
    if (isAnagram(s1, s2))
        printf("\"%s\" and \"%s\" are anagrams", s1, s2);
    else
        printf("\"%s\" and \"%s\" are not anagrams", s1, s2);
    return 0;
}
```

Explanation:

Pointers p1 and p2 traverse s1 and s2 separately, using each character's numeric (ASCII) value as an index into a frequency array — incrementing it while scanning s1 and decrementing it while scanning s2. If both strings

contain exactly the same characters with the same frequencies, every count returns to zero, confirming they are anagrams.

Sample Output:

```
"listen" and "silent" are anagrams
```

Problem 12: Write a C program to find the GCD (HCF) of two numbers using pointers and a function.

Program:

```
#include <stdio.h>
void findGCD(int a, int b, int *result) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    *result = a;
}
int main() {
    int a = 36, b = 60, gcd;
    findGCD(a, b, &gcd);
    printf("GCD of %d and %d = %d", a, b, gcd);
    return 0;
}
```

Explanation:

The function computes the GCD internally using the Euclidean algorithm, but instead of returning the answer with a return statement, it writes the result directly into the caller's gcd variable through the pointer parameter result (*result = a;) — a common pattern when a function's return value is already used for something else, such as a success/failure status.

Sample Output:

```
GCD of 36 and 60 = 12
```

Problem 13: Write a C program to implement a simple queue (enqueue and dequeue) using an array and pointers.

Program:

```
#include <stdio.h>
#define MAX 5
int queue[MAX];
int *front = queue, *rear = queue - 1;
int count = 0;
void enqueue(int value) {
    if (count == MAX) {
        printf("Queue overflow\n");
        return;
    }
    rear++;
    *rear = value;
```

```
    count++;
}
int dequeue() {
    if (count == 0) {
        printf("Queue underflow\n");
        return -1;
    }
    int value = *front;
    front++;
    count--;
    return value;
}
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    printf("Dequeued: %d\n", dequeue());
    printf("Dequeued: %d\n", dequeue());
    enqueue(40);
    printf("Dequeued: %d\n", dequeue());
    printf("Dequeued: %d\n", dequeue());
    return 0;
}
```

Explanation:

Two pointers manage the queue: rear advances and writes a new value whenever enqueue() is called, while front advances and reads the oldest remaining value whenever dequeue() is called. Because front always trails rear, elements are removed in the same order they were added — first in, first out — implemented entirely through pointer movement rather than shifting array elements.

Sample Output:

```
Dequeued: 10
Dequeued: 20
Dequeued: 30
Dequeued: 40
```

Problem 14: Write a C program to demonstrate a pointer to an array, and explain how it differs from an array of pointers.

Program:

```
#include <stdio.h>
int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int (*p)[5];      // p is a single pointer to an array of 5 integers
    p = &arr;
    printf("Elements accessed via pointer to array:\n");
    for (int i = 0; i < 5; i++)
        printf("%d ", (*p)[i]);
    printf("\n");
    return 0;
}
```

Explanation:

The declaration int (*p)[5]; makes p a single pointer that holds the address of an entire array of 5 integers, so p = &arr; is valid and *p yields the whole array back, allowing (*p)[i] to access individual elements. This is fundamentally different from int *p[5];, which instead declares an array of 5 separate pointers, each capable of pointing to a different, independent integer or block of memory.

Sample Output:

```
Elements accessed via pointer to array:
10 20 30 40 50
```

Problem 15: Write a C program to demonstrate the difference between a pointer to a constant (const int *p) and a constant pointer (int * const p).

Program:

```c
#include <stdio.h>
int main() {
    int a = 10, b = 20;
    const int *p1 = &a;   // pointer to a constant: value cannot change via p1
    int *const p2 = &a;   // constant pointer: address cannot change

    // *p1 = 15;   // NOT allowed -- would cause a compile error
    p1 = &b;          // allowed -- p1 can be redirected to another variable
    printf("p1 now points to b = %d\n", *p1);

    *p2 = 99;         // allowed -- the value at the fixed address can change
    // p2 = &b;    // NOT allowed -- would cause a compile error
    printf("a is now = %d (changed via p2)\n", a);

    return 0;
}
```

Explanation:

With const int *p1, the data p1 points to is treated as read-only, so *p1 = 15; would fail to compile, but p1 itself can still be reassigned to a different address (p1 = &b;). With int *const p2, it is the pointer itself that is fixed — p2 must always point to a, so p2 = &b; would fail to compile — but the value stored at that fixed address can still be changed through *p2 = 99;. Confusing these two declarations is one of the most frequently asked viva questions on pointers.

Sample Output:

```
p1 now points to b = 20
a is now = 99 (changed via p2)
```