C programming practice questions : Arrays , Strings, and Pointers

What is Array?
Array is a linear data structure where all elements are arranged sequentially. It is a collection of elements of same data type stored at contiguous memory locations.

For simplicity, we can think of an array as a flight of stairs where on each step is placed a value (let's say one of your friends). Here, you can identify the location of any of your friends by simply knowing the count of the step they are on. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array). The base value is index 0 and the difference between the two indexes is the offset. Is the array always of a fixed size? Arrays at core are of fixed size only, but most of the languages provide dynamic sized arrays using the underlying fixed sized arrays. For example, vector in C++, ArrayList in Java and list in Python. In C language, the array has a fixed size meaning once the size is given to it, it cannot be changed i.e. you can't shrink it nor can you expand it. Important Points to Note about Arrays Array is a fundamental data structure and used to implement other data structures like stack, queue, dequeue and heap. The main advantages of using array over other data structures are cache friendliness and random access memory. Please refer Getting Started with Array Data Structure for more details.

1. Based on Dimensions: One-Dimensional (1D) Arrays: These arrays store elements in a single, linear sequence, similar to a list. Each element is accessed using a single index. Example: int numbers[5]; (declares an array named numbers that can hold 5 integers).

Multi-Dimensional Arrays: These arrays store elements in multiple dimensions, allowing for representation of data in a tabular or matrix-like form.

Two-Dimensional (2D) Arrays: Represent data in rows and columns, like a grid or a spreadsheet. Elements are accessed using two indices (row and column). Example: int matrix[3][4]; (declares a 2D array with 3 rows and 4 columns).

Three-Dimensional (3D) Arrays: Extend the concept to include a third dimension, often visualized as layers or depth. Elements are accessed using three indices. Example: int cube[2][3][4]; (declares a 3D array**).**

Jagged Arrays (Arrays of Arrays): A specific type of multi-dimensional array where each inner array can have a different length. This means the rows are not necessarily uniform in size.

2. Based on Size Flexibility: Fixed-Size Arrays: The size of the array is determined at the time of declaration and cannot be changed during program execution.

Dynamic Arrays: The size of the array can be modified (increased or decreased) during program execution, allowing for more flexible memory management.

Q1: Write a C program to store and print the first 10 natural numbers using an array

```
n=int(input())
for i in range(1,n+1):
    print(i)
```

In c

```
#include <stdio.h>

int main() {
    int a;
    printf("enter a number  ");
    scanf("%d",&a);
    for(int i=1;i<=a;i++){
        printf("%d \n",i);
    }

    return 0;
}
```

Q2: Explain the difference between declaring int arr[10]; and int *arr; in terms of memory allocation.

When you writee

```
int arr[10];
```

you are telling the compiler: *“Give me space right now for exactly 10 integers, side by side, in one continuous block.”*
The compiler immediately reserves that memory. If this line is inside a function, those 10 integers live on the stack. If it’s outside all functions (global) or marked `static`, they live in a fixed data section of your program.

From the moment the program runs, `arr` already has a home for its 10 integers. The size is known at compile time, so the compiler can do things like `sizeof(arr)` and get the exact number of bytes (10 times the size of an `int`). The name `arr` is not a variable in the normal sense — it’s more like a label for the block of memory. You cannot make `arr` point somewhere else, and its size will never change.

When you write

```
int *arr;
```

the situation is very different. This time, you’re only asking for space to store one thing: the address of an integer. That’s all. No integers themselves are created yet. It’s like saying, *“Give me a slip of paper where I can write down the location of some integers — I’ll figure out where later.”*

At this moment, the pointer `arr` doesn’t know where to point. It might hold garbage until you give it a proper address. You can assign it the address of an existing array, or you can use something like `malloc` to ask the operating system for a block of integers on the heap. If you choose `malloc(10 * sizeof(int))`, now `arr` will hold the address of that new heap block, but the pointer itself still lives wherever it was declared (on the stack if local, in the data segment if global).

This is why `int arr[10]` is a fixed, pre-built set of chairs — the space is already there and arranged neatly. `int *arr` is just a signpost — you can point it to any row of chairs you build later, and the row can be longer, shorter, or even moved entirely, but you must take care of creating and destroying it yourself.

: Q1: Write a program to find the maximum and minimum elements in an array of n integers.

import math
n = int(input())

```
arr = [0] * n
mi = math.inf
mx = -math.inf
for i in range(n):
    arr[i] = int(input("Enter a number: "))
    if arr[i] > mx:
        mx = arr[i]
    if arr[i] < mi:
        mi = arr[i]
print(mx, mi)
```

Write a program to reverse the elements of a 1D array in place

```
a=[1,2,3]
print(a[::-1])
```

Write a program to rotate an array by k positions without using an extra array.

```
def reverse(arr, start, end):
    while start < end:
        arr[start], arr[end] = arr[end], arr[start]
        start += 1
        end -= 1
n = int(input("Enter number of elements: "))
arr = [int(input()) for _ in range(n)]
k = int(input("Enter rotation positions: "))
k %= n
reverse(arr, 0, n - k - 1)
reverse(arr, n - k, n - 1)
reverse(arr, 0, n - 1)
print("Rotated array:", arr)
```

: Implement a binary search in a sorted array using recursion

```
def binary_search(arr, low, high, target):
    if low > high:
        return -1

    mid = (low + high) // 2

    if arr[mid] == target:
        return mid
    elif arr[mid] > target:
        return binary_search(arr, low, mid - 1, target)
    else:
        return binary_search(arr, mid + 1, high, target)
n = int(input("Enter number of elements: "))
arr = [int(input()) for _ in range(n)]
arr.sort()
target = int(input("Enter the element to search: "))
index = binary_search(arr, 0, n - 1, target)
if index != -1:
    print(f"Element found at index {index}")
else:
    print("Element not found")
```

Write a program to add two 3×3 matrices.

```
matrix1 = []
matrix2 = []

print("Enter elements of first 3x3 matrix:")
for i in range(3):
    row = []
    for j in range(3):
        row.append(int(input(f"Enter element [{i}][{j}]: ")))
    matrix1.append(row)

print("\nEnter elements of second 3x3 matrix:")
for i in range(3):
    row = []
    for j in range(3):
        row.append(int(input(f"Enter element [{i}][{j}]: ")))
    matrix2.append(row)
result = []
for i in range(3):
    row = []
    for j in range(3):
        row.append(matrix1[i][j] + matrix2[i][j])
    result.append(row)
print("\nSum of the two matrices:")
for row in result:
    print(row)
```

Write a program to search for a specific element in a 2D array.

```
rows = int(input("Enter number of rows: "))
cols = int(input("Enter number of columns: "))
matrix = []
print("Enter the elements row-wise:")
for i in range(rows):
    row = list(map(int, input().split()))
    if len(row) != cols:
        print("Please enter exactly", cols, "elements for this row.")
        exit()
    matrix.append(row)
target = int(input("Enter the element to search: "))
found = False
for i in range(rows):
    for j in range(cols):
        if matrix[i][j] == target:
            print(f"Element {target} found at position ({i}, {j})")
            found = True

if not found:
    print(f"Element {target} not found in the array.")
```

: Implement matrix multiplication for two matrices of compatible dimensions.

```
r1 = int(input("Enter rows for first matrix: "))
c1 = int(input("Enter columns for first matrix: "))

r2 = int(input("Enter rows for second matrix: "))
c2 = int(input("Enter columns for second matrix: "))

if c1 != r2:
    print("Matrix multiplication not possible. (Columns of first != Rows of second)")
    exit()
print("Enter elements of first matrix row-wise:")
A = []
for i in range(r1):
    A.append(list(map(int, input().split())))

print("Enter elements of second matrix row-wise:")
B = []
for i in range(r2):
    B.append(list(map(int, input().split())))
result = [[0] * c2 for _ in range(r1)]

for i in range(r1):
    for j in range(c2):
        for k in range(c1):
            result[i][j] += A[i][k] * B[k][j]

print("Resultant Matrix:")
for row in result:
    print(*row)
```

Write a program to find the transpose of a matrix in place.

```
n = int(input("Enter size of square matrix (n x n): "))

print("Enter the elements row-wise:")
matrix = []
for _ in range(n):
    matrix.append(list(map(int, input().split())))

for i in range(n):
    for j in range(i + 1, n):
        matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]

print("Transposed matrix:")
for row in matrix:
    print(*row)
```

Pointers
In programming, a pointer is a variable that stores the memory address of another variable, rather than holding a direct value itself. Pointers allow efficient access and manipulation of data in computer memory, which is essential for system-level programming, dynamic memory allocation, and improving performance. Key operations include using the & operator to get the address of a variable and the * operator (dereferencing) to access the value at that address.

How Pointers Work

Memory Addresses: Every variable in a program is stored at a specific location (address) in the computer's memory.

Storing an Address: A pointer variable holds this memory address.

Accessing Data:

The & (address-of) operator returns the memory address of a variable.

The * (dereference) operator allows you to access the value stored at the memory address that the pointer holds.

C

```
#include <stdio.h>

int main() {

int myVar = 10; // Declare an integer variable

int *ptr; // Declare a pointer to an integer

ptr = &myVar; // ptr now stores the memory address of myVar

printf("Value of myVar: %d\n", myVar); // Output: 10

printf("Address of myVar: %p\n", &myVar); // Output: Memory address of myVar

printf("Value of ptr (the address): %p\n", ptr); // Output: Same memory address as myVar

printf("Value at the address ptr points to: %d\n", *ptr); // Output: 10

return 0;
}
```

Common Use Cases

Dynamic Memory Allocation:

Pointers are used in languages like C and C++ for allocating and deallocating memory during program execution, allowing for flexible data structures.

Pass by Address:

Pointers enable functions to modify the original variables passed to them, which is more efficient for large data structures.

Data Structures:

They are fundamental for creating complex data structures such as linked lists and trees, where nodes are connected via memory addresses.

Arrays and Strings:

Pointers can provide efficient ways to access and manipulate large arrays and strings in memory.

Q1: Access and modify a variable using a pointer

```
#include <stdio.h>
int main() {
   int a = 10;
   int *p = &a;

   printf("Original value: %d\n", a);
   *p = 20;
   printf("Modified value: %d\n", a);

   return 0;
}
```

Q2: Assigning one pointer to another

```
#include <stdio.h>
int main() {
```

```c
    int x = 5;
    int *p1 = &x;
    int *p2;

    p2 = p1;

    printf("p1 value: %d\n", *p1);
    printf("p2 value: %d\n", *p2);

    *p2 = 15;
    printf("After modification: %d\n", x);

    return 0;
}
```

Q1: Traverse and print array

```c
#include <stdio.h>
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *p = arr;
    int n = 5;

    for(int i = 0; i < n; i++) {
        printf("%d ", *(p + i));
    }
    return 0;
}
```

Copy one array to another

```c
#include <stdio.h>
int main() {
    int src[] = {1, 2, 3, 4, 5};
    int dest[5];
    int *p1 = src, *p2 = dest;
    int n = 5;

    for(int i = 0; i < n; i++) {
        *(p2 + i) = *(p1 + i);
    }

    printf("Copied array: ");
    for(int i = 0; i < n; i++) {
        printf("%d ", dest[i]);
    }
    return 0;
}
```

Q1: Sum of array elements

```
#include <stdio.h>
int main() {
    int arr[] = {5, 10, 15, 20};
    int n = 4, sum = 0;
    int *p = arr;

    for(int i = 0; i < n; i++) {
        sum += *(p + i);
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

Q2: Reverse a string

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[] = "hello";
    char *p1 = str, *p2 = str + strlen(str) - 1;
    char temp;

    while(p1 < p2) {
        temp = *p1;
        *p1 = *p2;
        *p2 = temp;
        p1++;
        p2--;
    }
```

```c
    printf("Reversed: %s\n", str);
    return 0;
}
```

Q1: Count words in a string

```c
#include <stdio.h>
#include <ctype.h>
int main() {
    char str[] = "Pointers are powerful in C";
    char *p = str;
    int count = 0, inWord = 0;

    while(*p) {
        if(!isspace(*p) && !inWord) {
            count++;
            inWord = 1;
        } else if(isspace(*p)) {
            inWord = 0;
        }
        p++;
    }
    printf("Word count = %d\n", count);
    return 0;
}
```

Q2: Replace spaces with '-'

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

    while(*p) {
        if(*p == ' ')
            *p = '-';
        p++;
    }
    printf("Modified string: %s\n", str);
    return 0;
}
```

Q1: Access 2D array

```
#include <stdio.h>
int main() {
    int arr[2][3] = {{1,2,3},{4,5,6}};
    int *ptr[2] = {arr[0], arr[1]};

    for(int i = 0; i < 2; i++) {
        for(int j = 0; j < 3; j++) {
            printf("%d ", *(*(ptr + i) + j));
        }
        printf("\n");
    }
    return 0;
}
```

Q2: How `char **argv` works

```
#include <stdio.h>
int main(int argc, char **argv) {
    printf("Number of arguments: %d\n", argc);
    for(int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    return 0;
}
```

Dynamic Memory Allocation

Dynamic memory allocation is a programming technique that allows you to allocate memory space for variables during the execution of a program, rather than at compile time. This flexibility is crucial when the exact amount of memory needed isn't known beforehand or when data structures need to grow or shrink during runtime.

Here's a more detailed explanation:

Static vs. Dynamic Memory Allocation

Static Memory Allocation:

Memory is allocated during compile time, and the size of the allocated space remains fixed throughout the program's execution. This is suitable for fixed-size data structures or when the memory requirements are known in advance.

Dynamic Memory Allocation:

Memory is allocated during runtime as needed, and the size can be adjusted during the program's lifespan. This is beneficial when dealing with data whose size is not known beforehand or when memory needs to change during the program's operation.

How Dynamic Memory Allocation Works

1. Requesting Memory:

You use specific functions (like malloc, calloc, or new in C/C++) to request memory from the operating system during program execution.

2. Receiving a Pointer:

These functions return a pointer (an address) to the allocated memory block.

3. Using the Memory:

You can then use this pointer to store and access data within the allocated memory space.

4. Releasing Memory:

It's essential to release the allocated memory using functions like free or delete when it's no longer needed to prevent memory leaks.

Benefits of Dynamic Memory Allocation

Flexibility: Accommodates changing memory requirements.

Efficiency: Allocates memory only when needed, minimizing waste.

Dynamic Data Structures: Enables the use of dynamic data structures like linked lists, trees, etc.

Optimized Resource Usage: Better utilization of available memory resources.

Potential Drawbacks

Memory Leaks: If allocated memory is not freed when no longer needed, it can lead to memory leaks.

Overhead: Dynamic allocation can have some runtime overhead associated with it.

Fragmentation: Repeated allocation and deallocation can lead to memory fragmentation, potentially impacting performance.

Q1: Dynamically create 2D array

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int rows = 3, cols = 4;
   int **arr = (int **)malloc(rows * sizeof(int *));
   for(int i = 0; i < rows; i++)
      arr[i] = (int *)malloc(cols * sizeof(int));
   for(int i = 0; i < rows; i++)
      for(int j = 0; j < cols; j++)
         arr[i][j] = i + j;

   for(int i = 0; i < rows; i++) {
      for(int j = 0; j < cols; j++)
         printf("%d ", arr[i][j]);
      printf("\n");
   }

   for(int i = 0; i < rows; i++)
      free(arr[i]);
   free(arr);

   return 0;
}
```

Q2: Swap two strings

```
#include <stdio.h>
void swapStrings(char **s1, char **s2) {
   char *temp = *s1;
   *s1 = *s2;
   *s2 = temp;
}
```

```
int main() {
    char *str1 = "Hello";
    char *str2 = "World";

    printf("Before: %s, %s\n", str1, str2);
    swapStrings(&str1, &str2);
    printf("After: %s, %s\n", str1, str2);

    return 0;
}
```

Q1: malloc example

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n;
    printf("Enter n: ");
    scanf("%d", &n);

    int *arr = (int *)malloc(n * sizeof(int));
    for(int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("You entered: ");
    for(int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    free(arr);
    return 0;
}
```

calloc + realloc

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n;
    printf("Enter n: ");
    scanf("%d", &n);
    int *arr = (int *)calloc(n, sizeof(int));
    for(int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    arr = (int *)realloc(arr, (n + 5) * sizeof(int));
    printf("Enter 5 more elements:\n");
    for(int i = n; i < n + 5; i++)
        scanf("%d", &arr[i]);

    printf("Updated array: ");
    for(int i = 0; i < n + 5; i++)
        printf("%d ", arr[i]);

    free(arr);
    return 0;
}
```

Advanced: Matrix addition with dynamic memory

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int r, c;
    printf("Enter rows and cols: ");
    scanf("%d %d", &r, &c);

    int **A = (int **)malloc(r * sizeof(int *));
    int **B = (int **)malloc(r * sizeof(int *));
    int **C = (int **)malloc(r * sizeof(int *));
    for(int i = 0; i < r; i++) {
        A[i] = (int *)malloc(c * sizeof(int));
        B[i] = (int *)malloc(c * sizeof(int));
        C[i] = (int *)malloc(c * sizeof(int));
    }

    printf("Enter A:\n");
    for(int i = 0; i < r; i++)
        for(int j = 0; j < c; j++)
            scanf("%d", &A[i][j]);

    printf("Enter B:\n");
    for(int i = 0; i < r; i++)
        for(int j = 0; j < c; j++)
            scanf("%d", &B[i][j]);

    for(int i = 0; i < r; i++)
        for(int j = 0; j < c; j++)
            C[i][j] = A[i][j] + B[i][j];

    printf("Result:\n");
    for(int i = 0; i < r; i++) {
        for(int j = 0; j < c; j++)
            printf("%d ", C[i][j]);
        printf("\n");
    }

    for(int i = 0; i < r; i++) {
        free(A[i]); free(B[i]); free(C[i]);
    }
    free(A); free(B); free(C);
    return 0;
}
```

Advanced: Store multiple strings dynamically

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
    int n;
    printf("Enter number of strings: ");
    scanf("%d", &n);

    char **arr = (char **)malloc(n * sizeof(char *));
    char buffer[100];

    for(int i = 0; i < n; i++) {
        printf("Enter string %d: ", i+1);
        scanf("%s", buffer);
        arr[i] = (char *)malloc((strlen(buffer)+1) * sizeof(char));
        strcpy(arr[i], buffer);
    }

    printf("Stored strings:\n");
    for(int i = 0; i < n; i++)
        printf("%s\n", arr[i]);
    for(int i = 0; i < n; i++)
        free(arr[i]);
    free(arr);

    return 0;
}
```

strings:

In C programming, a string is fundamentally a one-dimensional array of characters that is terminated by a special null character, represented as \0. This null character signifies the end of the string, allowing functions to determine its length and process it correctly.

Key characteristics of C strings:

Character Arrays:

Strings are implemented as arrays of char data type.

Null Termination:

Every C string must end with a null character (\0). This character is automatically added by the compiler when you use string literals (e.g., "Hello"). If you initialize a string character by character, you must explicitly add \0.

Memory Allocation:

Each character in a string occupies one byte of memory. The null terminator also occupies one byte, so a string of n characters requires n + 1 bytes of storage.

Declaration and Initialization:

String Literal: char str[] = "Hello"; (compiler automatically adds \0 and determines size).

Fixed Size Array: char str[10]; (reserves space for 9 characters + \0).

Character by Character: char str[] = {'H', 'e', 'l', 'l', 'o', '\0'}; (explicitly includes \0).

Common String Operations and Functions:

C provides a set of standard library functions (primarily in <string.h>) for manipulating strings, including:

strlen(): Calculates the length of a string (excluding the null terminator).

strcpy(): Copies one string to another.

strcat(): Concatenates (joins) two strings.

strcmp(): Compares two strings lexicographically.

strstr(): Finds the first occurrence of a substring within a string.

Strings in C: Q1: Write a program to count the number of vowels in a given string without using library functions.

```
n=input()
c=0
for i in n:
    if i =='a' or i=='e' or i=='i' or i=='o' or i=='u':
        c+=1
print(c)
```

Q2: Explain the difference between a character array (char str[]) and a pointer to char (char *str).

A character array (`char str[]`) is an actual block of memory reserved to hold characters. When you write `char str[] = "Hello";`, the compiler allocates space for each character of `"Hello"` plus the null terminator. The contents of this array can usually be modified (e.g., changing `'H'` to `'h'`). However, since it is an array, its name (`str`) always refers to the same fixed memory location — you cannot make `str` point somewhere else after it has been declared.

A pointer to char (`char *str`), on the other hand, is just a variable that stores the address of a character (or the first character of a string). For example, if you write `char *str = "Hello";`, the string `"Hello"` is stored in read-only memory, and the pointer `str` simply points to it. The pointer itself can later be reassigned to point to another string or memory location. But if it points to a string literal, trying to modify the characters will cause undefined behavior, because string literals are not meant to be changed.

: Q1: Write a program to check whether a given string is a palindrome using strlen() and strcmp().

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

int main() {
    char str[100], rev[100];
    int len, i;

    printf("Enter a string: ");
    scanf("%s", str);

    len = strlen(str);
    for (i = 0; i < len; i++) {
        rev[i] = str[len - i - 1];
    }
    rev[len] = '\0';
    if (strcmp(str, rev) == 0) {
        printf("The string is a palindrome.\n");
    } else {
        printf("The string is not a palindrome.\n");
    }

    return 0;
}
```

Q2: Write a program to concatenate two strings using strcat().

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

int main() {
    char str1[100], str2[50];
    printf("Enter first string: ");
    gets(str1);

    printf("Enter second string: ");
    gets(str2);

    strcat(str1, str2);

    printf("Concatenated string: %s\n", str1);

    return 0;
}
```

Advanced: Implement your own strlen(), strcpy(), and strcmp() functions without using the string.h library.

```
#include <stdio.h>

int my_strlen(char *s) {
    int len = 0;
    while (s[len] != '\0') {
        len++;
    }
    return len;
}

void my_strcpy(char *dest, char *src) {
    int i = 0;
    while ((dest[i] = src[i]) != '\0') {
        i++;
    }
}

int my_strcmp(char *s1, char *s2) {
    int i = 0;
    while (s1[i] != '\0' && s2[i] != '\0') {
        if (s1[i] != s2[i])
            return s1[i] - s2[i];
        i++;
    }
    return s1[i] - s2[i];
}

int main() {
    char a[50] = "hello", b[50] = "world", c[50];

    printf("Length of '%s' = %d\n", a, my_strlen(a));

    my_strcpy(c, a);
    printf("Copied string: %s\n", c);

    printf("Comparing '%s' and '%s' = %d\n", a, b, my_strcmp(a, b));

    return 0;
}
```

Write a program to sort an array of 5 strings in alphabetical order.

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

int main() {
    char str[5][50], temp[50];
    int i, j;

    printf("Enter 5 strings:\n");
    for (i = 0; i < 5; i++)
        gets(str[i]);

    for (i = 0; i < 5 - 1; i++) {
        for (j = i + 1; j < 5; j++) {
            if (strcmp(str[i], str[j]) > 0) {
                strcpy(temp, str[i]);
                strcpy(str[i], str[j]);
                strcpy(str[j], temp);
            }
        }
    }

    printf("\nStrings in alphabetical order:\n");
    for (i = 0; i < 5; i++)
```

```
        printf("%s\n", str[i]);

    return 0;
}
```

: Write a program to search for a name in an array of names using strcmp().

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

int main() {
    char names[5][50] = {"Ravi", "Anita", "Praneeth", "Sita", "Rahul"};
    char search[50];
    int i, found = 0;

    printf("Enter name to search: ");
    gets(search);

    for (i = 0; i < 5; i++) {
        if (strcmp(names[i], search) == 0) {
            found = 1;
            break;
        }
    }
    if (found)
        printf("Name '%s' found at position %d\n", search, i + 1);
    else
        printf("Name not found!\n");

    return 0;
}
```

Write a program to find the longest string in an array of strings.

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

int main() {
    char str[5][50];
    int i, maxLen = 0, index = 0;
    printf("Enter 5 strings:\n");
    for (i = 0; i < 5; i++)
        gets(str[i]);
    for (i = 0; i < 5; i++) {
        int len = strlen(str[i]);
        if (len > maxLen) {
            maxLen = len;
            index = i;
        }
    }
    printf("\nLongest string is: %s (Length = %d)\n", str[index], maxLen);
    return 0;
}
```