Arrays and Pointers Assignment

1. What does the code below refer to? Extend the code and demonstrate the use of ptr to access the contents of a 2D array.

```
int (*ptr)[4];
```

sol:

```
#include <stdio.h>
int main() {
   int arr[][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };
   int (*ptr)[4];
   int i;
   ptr = arr;
   printf("First row: ");
   for ( i = 0; i < 4; i++) {
      printf("%d ", (*ptr)[i]);
   }
 printf("\n");
   ptr++;
   printf("Second row: ");
   for ( i = 0; i < 4; i++) {
      printf("%d ", (*ptr)[i]);
   }
   printf("\n");
   ptr++;
   printf("Third row: ");
   for ( i = 0; i < 4; i++) {
      printf("%d ", (*ptr)[i]);
   }
   printf("\n");
   return 0;
}
```

```
user57@trainux01:~/Batch17OCT2024$ vi arr_point1.c
user57@trainux01:~/Batch17OCT2024$ gcc arr_point1.c
user57@trainux01:~/Batch17OCT2024$ ./a.out
First row: 1 2 3 4
Second row: 5 6 7 8
Third row: 9 10 11 12
```

2. Refer the code in “array_ptr_simple_char.c”. Implement the missing functionality in the code marked with TBD1, TBD2.....

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

   int arr[][3] = {1, 2, 3, 4, 5, 6};

   printf("\n 1: %d %d", arr[1][0], arr[0][2]);

   int *ptr = arr[1];

   printf("\n 2:  %d %d %d", *(ptr+0), *(ptr+1), *(ptr+2));

   int *ptr2[3]; // Array of 3 pointers

   ptr2[0] = &arr[0][0];

   ptr2[1] = &arr[0][1];

   ptr2[2] = &arr[0][2];

   printf("\n 3:  %d %d %d", *ptr2[0], *ptr2[1], *ptr2[2]);

   int (*ptr3)[3]; /

   ptr3 = arr; // Point to the first row

   printf("\n 4: %d %d %d", (*ptr3)[0], (*ptr3)[1], (*ptr3)[2]);

   ptr3++; // Move to the second row

   printf("\n 4: %d %d %d", (*ptr3)[0], (*ptr3)[1], (*ptr3)[2]);

   return 0;

}
```

```
user57@trainux01:~/Batch17OCT2024$ vi arr_point2.c
user57@trainux01:~/Batch17OCT2024$ gcc arr_point2.c
user57@trainux01:~/Batch17OCT2024$ ./a.out

 1: 4 3
 2:  4 5 6
 3:  1 2 3
 4: 1 2 3
 4: 4 5 6user57@trainux01:~/Batch17OCT2024$
```

3. Refer the code snippet below. Implement the function search_insert() as mentioned in the code.

```
#include <stdio.h>
#include <string.h>
#define MAX 80
#define SUCCESS 1
#define FAILURE 0
// Function prototype
int search_insert(char name[], char search_char);
int i,j;
int main() {
   char name[MAX] = "ABC";  // Initial string
   char search_char = 'B';   // Character to search
   int ret = search_insert(name, search_char);
   if (ret == SUCCESS) {
      printf("Updated string: %s\n", name);
   } else {
      printf("Character not found or insertion failed.\n");
   }
   return 0;
}
int search_insert(char name[], char search_char) {
   int len = strlen(name);  // Get the length of the string
   for (i = 0; i < len; i++) {
      if (name[i] == search_char) {
         // If space is available to insert '_', we shift characters
         if (len + 1 < MAX) {  // Ensure there is space for the new character
            // Shift the characters starting from the position after search_char
            for ( j = len; j >= i + 1; j--) {
               name[j + 1] = name[j];
            }
```

```
            // Insert the underscore after the search_char
            name[i + 1] = '_';
            name[len + 1] = '\0';  // Null-terminate the string
            return SUCCESS;
        } else {
            // If there is no space for insertion
            return FAILURE;
        }
    }
  }
 return FAILURE;  // If search_char was not found
}
```

Output:

```
user57@trainux01:~/Batch17OCT2024$ vi arr_point3.c
user57@trainux01:~/Batch17OCT2024$ gcc arr_point3.c
user57@trainux01:~/Batch17OCT2024$ ./a.out
Updated string: AB_C
```

4. Refer the program “array_ptr_repr_partial.c”. Implement the functions below which are yet to be implemented in code.

int func1(int (*ptr)[3]); // pointer to array, second dimension is explicitly specified

int func2(int **ptr); // double pointer, using an auxiliary array of pointers

sol:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 3
int func1(int (*ptr)[3]);
int func2(int **ptr);
int i;
int main() {
int arr1[2][3] = {{1, 2, 3}, {4, 5, 6}};
   printf("func1 output: %d\n", func1(arr1));
```

```
   int **arr2;
   arr2 = (int **)malloc(2 * sizeof(int *)); // Allocate space for 2 rows
   for ( i = 0; i < 2; i++) {
      arr2[i] = (int *)malloc(3 * sizeof(int));
   }
   arr2[0][0] = 1; arr2[0][1] = 2; arr2[0][2] = 3;
   arr2[1][0] = 4; arr2[1][1] = 5; arr2[1][2] = 6;
  printf("func2 output: %d\n", func2(arr2));
   for ( i = 0; i < 2; i++) {
      free(arr2[i]);
   }
   free(arr2);
return 0;
}
int func1(int (*ptr)[3]) {
   return ptr[0][0];
}
int func2(int **ptr) {
return ptr[1][1];
}
```

```
user57@trainux01:~/Batch17OCT2024$ vi arr_point4.c
user57@trainux01:~/Batch17OCT2024$ gcc arr_point4.c
user57@trainux01:~/Batch17OCT2024$ ./a.out
func1 output: 1
func2 output: 5
```

5. Refer the program “array_dbl_pointers_function_partial.c”. Implement the missing functionality in the code marked with TBD1, TBD2…..

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

```
#include <stdlib.h>

void func1(short mat[][3]);
void func2(short (*ptr)[3]);
void func3(short *mat);
void func4(short **mat);
void func5(short *ptr[3]);
int main()
{
   short mat[3][3], i, j;
   // Initialize the matrix with values
   for(i = 0 ; i < 3 ; i++)
      for(j = 0 ; j < 3 ; j++)
      {
         mat[i][j] = i*10 + j;
      }
   // Display the initialized data
   printf(" Initialized data to: ");
   for(i = 0 ; i < 3 ; i++)
   {
      printf("\n");
      for(j = 0 ; j < 3 ; j++)
      {
         printf("%5.2d", mat[i][j]);
      }
   }
   printf("\n");
   func1(mat);
   func2(mat);
   func3(mat);
   func4(mat);
```

```
   func5(mat);
    return 0;
}

/*
Method #1 (No tricks, just an array with empty first dimension)
================================================================
You don't have to specify the first dimension!
*/

/*PROPER METHOD*/
void func1(short mat[][3])
{
   register short i, j;
   printf(" Declare as matrix, explicitly specify second dimension: ");
   for(i = 0 ; i < 3 ; i++)
   {
      printf("\n");
      for(j = 0 ; j < 3 ; j++)
      {
         // Display the element at mat[i][j]
         printf("%5.2d", mat[i][j]);
      }
   }
   printf("\n");
}
/*
Method #2 (pointer to array, second dimension is explicitly specified)
=====================================================================
*/
void func2(short (*mat)[3])
```

```
{
   register short i, j;

   printf(" Declare as pointer to column, explicitly specify 2nd dim: ");
   for(i = 0 ; i < 3 ; i++)
   {
      printf("\n");
      for(j = 0 ; j < 3 ; j++)
      {
         // Display the element at mat[i][j]
         printf("%5.2d", mat[i][j]);
      }
   }
   printf("\n");
}

/*
Method #3 (Using a single pointer, the array is "flattened")
============================================================
With this method you can create general-purpose routines.
The dimensions doesn't appear in any declaration, so you
can add them to the formal argument list.
The manual array indexing will probably slow down execution.
*/
void func3(short *mat)
{
   register short i, j;
   printf(" Declare as single-pointer, manual offset computation: ");
   for(i = 0 ; i < 3 ; i++)
   {
      printf("\n");
```

```
      for(j = 0 ; j < 3 ; j++)
      {
         // Calculate the element's offset in the flattened array
         printf("%5.2d", *(mat + i * 3 + j));
      }
   }
   printf("\n");
}

/*
Method #4 (double pointer, using an auxiliary array of pointers)
================================================================
With this method you can create general-purpose routines,
if you allocate "index" at run-time.
Add the dimensions to the formal argument list.
*/
void func4(short **mat)
{
   short i, j, *index[3];
  // Initialize the index array to point to each row of mat
   for (i = 0 ; i < 3 ; i++)
   {
      index[i] = mat[i];  // Pointing index[i] to mat[i], each row
   }
  printf(" Declare as double-pointer, use auxiliary pointer array: ");
   for(i = 0 ; i < 3 ; i++)
   {
      printf("\n");
      for(j = 0 ; j < 3 ; j++)
      {
         // Access elements using the auxiliary array of pointers
```

```
            printf("%5.2d", *(index[i] + j));
        }
    }
    printf("\n");
}
/*
Method #5 (single pointer, using an auxiliary array of pointers)
================================================================
*/
void func5(short *mat[3])
{
    short i, j, *index[3];
    // Initialize the index array to point to each row of mat
    for (i = 0 ; i < 3 ; i++)
    {
        index[i] = mat[i];  // Pointing index[i] to mat[i], each row
    }
    printf(" Declare as single-pointer, use auxiliary pointer array: ");
    for(i = 0 ; i < 3 ; i++)
    {
        printf("\n");
        for(j = 0 ; j < 3 ; j++)
        {
            // Access elements using the auxiliary array of pointers
            printf("%5.2d", *(index[i] + j));
        }
    }
    printf("\n");
}
```

6. Refer the program "pointer_example.c". Fix the warning issue.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   char arr[] = "ABC";
   char *ptr = arr;  //  pointing to the first character
   printf("%p %p", (void*)arr, (void*)ptr);  // Casting to void* for proper pointer printing
   printf("\n 1 %c %c", *ptr, *(ptr + 1));  // Dereferencing pointer to print characters

  // msg is an array of strings, each of size 5 (to hold 4 chars + null terminator)
   char msg[][5] = {"AB", "gh", "er"};

// ptr2 should point to an array of 5 characters (not 2)
   char (*ptr2)[5];

   ptr2 = &msg[0];  // Pointing to the first string in msg
   printf("\n %p %p", (void*)msg, (void*)ptr2);  // Casting to void* for proper pointer printing
   printf("\n 2 %c %c", (*ptr2)[0], (*ptr2)[1]);  // Dereferencing to print characters of the first string

   return 0;
}
```

7. Consider an array of strings as below.

```
char arr[][10]={"Word", "Excel", "PowerPoint", "Pdf", "Paint"};
```

a. Implement a function read_displaystring() to read a row index from the user, access the string, store in a char * variable and using this, traverse every alternate character in the string and display in console.

```
void read_displaystring(char *arr[][10], int row);
```

b. Reverse the string read at the index in a) using a function of prototype as below. Caller to read the returned string and display the reversed string. [Ensure that the input source array is not corrupted and remaining elements are intact]

```
char *reverse(char *arr[][10], int row)
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
char *reverse(char *arr[], int row) {
   char *str = arr[row];

   int length = strlen(str);
   char *reversed = (char *)malloc(length + 1);  // +1 for the null-terminator
   int i;
 // Reverse the string
   for ( i = 0; i < length; i++) {
      reversed[i] = str[length - 1 - i];
   }
   // Null-terminate the reversed string
   reversed[length] = '\0';
   return reversed;
}

int main() {
   char *arr[] = {"Word", "Excel", "PowerPoint", "Pdf", "Paint"};
   int row;

  // Take row index input from the user
   printf("Enter row index (0-4): ");
   scanf("%d", &row);

// Ensure row index is within valid bounds
```

```
    if (row < 0 || row > 4) {
        printf("Invalid row index!\n");
    } else {
        // Call reverse and get the reversed string
        char *reversed_str = reverse(arr, row);

       // Display the reversed string
        printf("Reversed string: %s\n", reversed_str);

        // Free the allocated memory
        free(reversed_str);
    }

    return 0;
}
```

```
user57@trainux01:~/Batch17OCT2024$ vi arr_pointer7.c
user57@trainux01:~/Batch17OCT2024$ gcc arr_pointer7.c
user57@trainux01:~/Batch17OCT2024$ ./a.out
Enter row index (0-4): 3
Reversed string: fdP
```