Write a c program to Create magic square matrix of order 3×3

```
#include <stdio.h>
void createMagicSquare(int n, int magicSquare[][n])
{
   // Initialize all elements of the magic square to 0
   for (int i = 0; i < n; i++)
   {
      for (int j = 0; j < n; j++)
      {
         magicSquare[i][j] = 0;
      }
   }
   int i = n / 2;     // Start position for 1
   int j = n - 1;
   for (int num = 1; num <= n * n;)
   {
      if (i == -1 && j == n)   // 3rd condition
      {
         j = n - 2;
         i = 0;
      }
      else
      {
         // Wrap around the rows and columns
         if (j == n)
            j = 0;
         if (i < 0)
            i = n - 1;
      }
      if (magicSquare[i][j])  // 2nd condition
      {
         j -= 2;
         i++;
         continue;
      }
```

```
      else
         magicSquare[i][j] = num++;  // Set the value
      j++;
      i--;
   }
}
void displayMagicSquare(int n, int magicSquare[][n])
{
   printf("Magic Square of size %dx%d:\n", n, n);
   for (int i = 0; i < n; i++)
   {
      for (int j = 0; j < n; j++)
      {
         printf("%3d ", magicSquare[i][j]);
      }
      printf("\n");
   }
}
int main()
{
   int n = 3;
   int magicSquare[n][n];
   createMagicSquare(n, magicSquare);
   displayMagicSquare(n, magicSquare);
   return 0;
}
#include <stdio.h
```

This line includes the standard input/output library, which provides functions like `printf` and `scanf`

```
Void createMagicSquare(int n, int magicSquare[][n])
{
   // Initialize all elements of the magic square to 0
   For (int I = 0; I < n; i++)
   {
      For (int j = 0; j < n; j++)
      {
```

```
        magicSquare[i][j] = 0;
      }
    }
```

This function `createMagicSquare` takes two parameters: `n` (the size of the magic square) and `magicSquare` (a 2D array representing the square). It initializes all elements of the magic square to 0.

```c
  Int I = n / 2;     // Start position for 1
  Int j = n – 1;
```

These two lines initialize `i` and `j` with the starting position for the number 1 in the magic square. The starting position is the middle column and the last row.

```c
  For (int num = 1; num <= n * n;)
  {
    If (I == -1 && j == n)   // 3rd condition
    {
      J = n – 2;
      I = 0;
    }
    Else
    {
      // Wrap around the rows and columns
      If (j == n)
        J = 0;
      If (I < 0)
        I = n – 1;
    }
```

This `for` loop runs until all the numbers from 1 to n^2 are placed in the magic square. It checks two conditions: if the current position is (-1, n) and wraps around to (0, n-2) and (n-1, 0) accordingly. The wrap-around is necessary to ensure the numbers are placed correctly in the magic square.

```c
    If (magicSquare[i][j])  // 2nd condition
    {
      J -= 2;
```

```
        I++;
        Continue;
      }
      Else
        magicSquare[i][j] = num++;  // Set the value

      j++;
      i--;
   }
```

This block of code checks if the current position is already occupied. If it is, it adjusts the position to move two columns back and one row down. The `continue` statement skips the rest of the loop and moves on to the next iteration. If the position is not occupied, it sets the current position to the current number (`num`) and increments `num` by 1. Then it adjusts the position by moving one column forward and one row up

```
Void displayMagicSquare(int n, int magicSquare[][n])
{
   Printf("Magic Square of size %dx%d:\n", n, n);
   For (int I = 0; I < n; i++)
   {
      For (int j = 0; j < n; j++)
      {
         Printf("%3d ", magicSquare[i][j]);
      }
      Printf("\n");
   }
}
```

The `displayMagicSquare` function takes the size `n` and the magic square array as parameters. It prints the magic square in a formatted manner using nested loops.

```c
Int main()
{
   Int n = 3;
   Int magicSquare[n][n];

   createMagicSquare(n, magicSquare);
   displayMagicSquare(n, magicSquare);
```

```
    return 0;
}
```

The `main` function initializes the size of the magic square (`n`) and declares a 2D array `magicSquare` to store the magic square values. It then calls the `createMagicSquare` function to generate the magic square and `displayMagicSquare` function to print it. Finally, it returns 0 to indicate successful program execution.

Write a c program to Create magic square matrix of order 5x5

```
#include <stdio.h>
Void createMagicSquare(int n, int magicSquare[][n]) {
    Int I, j;
    // Initialize all elements to 0
    For (I = 0; I < n; i++) {
        For (j = 0; j < n; j++) {
            magicSquare[i][j] = 0;
        }
    }
    // Set the position of number 1
    Int row = n / 2;
    Int col = n – 1;
    // Start placing numbers in the magic square
    For (int num = 1; num <= n * n; ) {
        If (row == -1 && col == n) {
            Row = 0;
            Col = n – 2;
        }
        Else {
            If (col == n)
                Col = 0;
            If (row < 0)
                Row = n – 1;
        }
        If (magicSquare[row][col]) {
            Row++;
            Col -= 2;
```

```
        Continue;
      }
      Else
        magicSquare[row][col] = num++;

      row--;
      col++;
    }
}
Void displayMagicSquare(int n, int magicSquare[][n]) {
   Printf("Magic Square of order %dx%d:\n", n, n);
   For (int I = 0; I < n; i++) {
      For (int j = 0; j < n; j++) {
        Printf("%3d ", magicSquare[i][j]);
      }
      Printf("\n");
   }
}
Int main() {
   Int n = 5;
   Int magicSquare[n][n];
   createMagicSquare(n, magicSquare);
   displayMagicSquare(n, magicSquare);
   return 0;
}
```

Explain

1. The code starts with the necessary header file `stdio.h`, which is required for input/output operations in C.
2. The `createMagicSquare` function is defined, which takes two arguments: `n` (the order of the magic square) and `magicSquare` (a 2D array to store the magic square).
3. Inside the `createMagicSquare` function, two variables `i` and `j` are declared for loop counters.
4. The nested `for` loops are used to initialize all elements of the `magicSquare` array to 0. It iterates over each row and column of the 2D array.
5. The position of the number 1 in the magic square is determined by setting `row` to `n / 2` and `col` to `n – 1`. This position is at the middle of the last column.
6. The next part is the main logic of creating the magic square. A `for` loop runs from `num = 1` to `num = n * n`, which represents the numbers to be placed in the magic square.
7. Inside the loop, there are several conditions to handle the wrapping and boundary cases of the magic square. If the row becomes -1 and column becomes n, it means the current position is outside the square, so it is wrapped around to the opposite side. Similarly, if the column becomes n, it is wrapped to the first column, and if the row becomes less than 0, it is wrapped to the last row.
8. If the current position in the magic square is already filled (non-zero), the row is incremented, and the column is decremented by 2 to move to the next available position.

9. Otherwise, the current number (`num`) is placed in the current position (`magicSquare[row][col]`), and `num` is incremented.
10. Finally, the `row` is decremented, and the `col` is incremented to move to the next position.
11. The `displayMagicSquare` function is defined, which takes the order `n` and the `magicSquare` array as arguments. It is responsible for printing the magic square in a readable format.
12. Inside the `displayMagicSquare` function, the order of the magic square is printed using `printf`.
13. Nested `for` loops are used to iterate over each row and column of the `magicSquare` array. The elements are printed using `printf` with a width specifier of 3, to align the numbers properly.
14. After printing each row, a newline character is printed to move to the next line.
15. In the `main` function, the order of the magic square `n` is set to 5.
16. An array `magicSquare` of size `n` by `n` is declared to store the magic square.
17. The `createMagicSquare` function is called, passing the order `n` and the `magicSquare` array to generate the magic square.
18. The `displayMagicSquare` function is called, passing the order `n` and the `magicSquare` array to print the generated magic square.
19. Finally, the `main` function returns 0 to indicate successful execution of the program.

Check whether entered matrix (order 3x3)is magic square or not ?

[8 1 6]
[3 5 7]
[4 9 2]

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

Bool isMagicSquare(int matrix[3][3]) {
  Int sum = 0;
  Int diagSum1 = 0;
  Int diagSum2 = 0;

  // Calculate the sum of the first row
  For (int I = 0; I < 3; i++) {
    Sum += matrix[0][i];
  }

  // Check the sum of each row and column
  For (int I = 1; I < 3; i++) {
    Int rowSum = 0;
    Int colSum = 0;

    For (int j = 0; j < 3; j++) {
      rowSum += matrix[i][j];
      colSum += matrix[j][i];
    }

    If (rowSum != sum || colSum != sum) {
      Return false;
    }
  }

  // Check the sum of the diagonals
  For (int I = 0; I < 3; i++) {
    diagSum1 += matrix[i][i];
    diagSum2 += matrix[i][2 – i];
  }

  If (diagSum1 != sum || diagSum2 != sum) {
    Return false;
  }

  Return true;
}

Int main() {
  Int matrix[3][3] = {{8, 1, 6}, {3, 5, 7}, {4, 9, 2}};
```

```
    Bool isMagic = isMagicSquare(matrix);

    If (isMagic) {
        Printf("The matrix is a magic square.\n");
    } else {
        Printf("The matrix is not a magic square.\n");
    }

    Return 0;
}
```

Explain

1. The code starts by including the necessary header files: `<stdio.h>` for input/output operations and `<stdbool.h>` for using the `bool` data type.
2. The `isMagicSquare` function is defined to check whether a given matrix is a magic square. It takes a 2D array `matrix` of integers as input and returns a boolean value (`true` or `false`).
3. Inside the `isMagicSquare` function, three variables are declared: `sum`, `diagSum1`, and `diagSum2`. These variables are used to calculate the expected sum for each row, column, and diagonal.
4. The first row of the matrix is traversed using a `for` loop, and the sum of its elements is calculated and stored in the `sum` variable. This sum will be used as the expected sum for each row, column, and diagonal.
5. Next, another `for` loop is used to check the sum of each row and column (excluding the first row). Two variables, `rowSum` and `colSum`, are used to calculate the sum of each row and column, respectively.
6. Inside the nested `for` loop, the `rowSum` variable is updated by adding the current element of the matrix at `matrix[i][j]`. Similarly, the `colSum` variable is updated by adding the current element at `matrix[j][i]`.
7. After calculating the sum of each row and column, an `if` condition is used to check whether `rowSum` or `colSum` is not equal to the expected `sum`. If any of these sums is different, it means the matrix is not a magic square, and the function returns `false`.
8. If all the row and column sums are equal to the expected `sum`, the function proceeds to check the sum of the diagonals. Two variables, `diagSum1` and `diagSum2`, are used to calculate the sum of the main diagonal and the anti-diagonal, respectively.
9. Another `for` loop is used to traverse the elements on the main diagonal and anti-diagonal. The `diagSum1` variable is updated by adding the element at `matrix[i][i]`, and the `diagSum2` variable is updated by adding the element at `matrix[i][2 - i]`.
10. Finally, an `if` condition is used to check whether `diagSum1` or `diagSum2` is not equal to the expected `sum`. If any of these sums is different, the matrix is not a magic square, and the function returns `false`.
11. If all the row sums, column sums, and diagonal sums are equal to the expected `sum`, the function returns `true`, indicating that the matrix is a magic square.
12. In the `main` function, a sample 3x3 matrix is declared and initialized with the values given in the example: `{{8, 1, 6}, {3, 5, 7}, {4, 9, 2}}`.
13. The `isMagicSquare` function is called with the `matrix` as an argument, and the returned boolean value is stored in the `isMagic` variable.
14. Finally, an `if-else` statement is used to print the appropriate message based on whether `isMagic` is `true` or `false`

Write a c program to Check whether entered matrix (order 5*5)is magic square or not ?

```
#include <stdio.h>


Int isMagicSquare(int matrix[][5]) {

    Int I, j;

    Int sum = 0;

    Const int order = 5;

    Int diagonalSum = 0;

    Int antiDiagonalSum = 0;


    // Calculate the sum of the first row (used as a reference)

    For (j = 0; j < order; j++) {
```

```
    Sum += matrix[0][j];
}

// Calculate the sum of each row and check if they match the reference sum
For (I = 1; I < order; i++) {
    Int rowSum = 0;
    For (j = 0; j < order; j++) {
        rowSum += matrix[i][j];
    }

    If (rowSum != sum) {
        Return 0; // Not a magic square
    }
}
// Calculate the sum of each column and check if they match the reference sum
For (j = 0; j < order; j++) {
    Int colSum = 0;
    For (I = 0; I < order; i++) {
        colSum += matrix[i][j];
    }
    If (colSum != sum) {
        Return 0; // Not a magic square
    }
}
// Calculate the sum of the diagonal and check if it matches the reference sum
For (I = 0; I < order; i++) {
    diagonalSum += matrix[i][i];
}
If (diagonalSum != sum) {
    Return 0; // Not a magic square
}
// Calculate the sum of the anti-diagonal and check if it matches the reference sum
For (I = 0; I < order; i++) {
    antiDiagonalSum += matrix[i][order – 1 – i];
}
```

```
    If (antiDiagonalSum != sum) {
        Return 0; // Not a magic square
    }

    Return 1; // Magic square
}

Int main() {
    Int matrix[5][5];
    Int I, j;

    Printf("Enter the elements of the matrix (5x5):\n");
    For (I = 0; I < 5; i++) {
        For (j = 0; j < 5; j++) {
            Scanf("%d", &matrix[i][j]);
        }
    }

    If (isMagicSquare(matrix)) {
        Printf("The entered matrix is a magic square.\n");
    } else {
        Printf("The entered matrix is not a magic square.\n");
    }

    Return 0;
}
```

1. The `isMagicSquare` function takes a 2D array `matrix` as input and returns an integer value. It checks whether the given matrix is a magic square or not.

2. It initializes variables `i`, `j`, `sum`, `order`, `diagonalSum`, and `antiDiagonalSum`.

3. The `sum` variable will store the sum of the first row, which will be used as a reference sum for comparisons.

4. The `order` variable is set to 5, representing the size of the matrix (5x5).

5. The `diagonalSum` and `antiDiagonalSum` variables will store the sum of the diagonal and anti-diagonal elements, respectively.

6. The function then calculates the sum of the first row (`matrix[0][j]`) and stores it in the `sum` variable.

7. It iterates over each row (starting from the second row) and calculates the sum of elements in each row. If any row's sum is not equal to the reference sum (`sum`), it immediately returns 0, indicating that the matrix is not a magic square.

8. It then iterates over each column and calculates the sum of elements in each column. If any column's sum is not equal to the reference sum (`sum`), it returns 0.

9. Next, it calculates the sum of the diagonal elements (`matrix[i][i]`) and checks if it matches the reference sum. If not, it returns 0.

10. Similarly, it calculates the sum of the anti-diagonal elements (`matrix[i][order – 1 – i]`) and checks if it matches the reference sum. If not, it returns 0.

11. If all the checks pass without returning 0, it means the matrix is a magic square, and the function returns 1.

12. In the `main` function, a 2D array `matrix` of size 5x5 is declared.

13. The user is prompted to enter the elements of the matrix. The nested loops iterate over each row and column, and `scanf` is used to read the elements from the user and store them in the matrix.

14. After the user enters the matrix, the `isMagicSquare` function is called with the `matrix` array as an argument.

15. If the return value from `isMagicSquare` is 1, it means the matrix is a magic square, so the program prints "The entered matrix is a magic square."

16. If the return value is not 1, it means the matrix is not a magic square, so the program prints "The entered matrix is not a magic square."

17. The program then terminates by returning 0.

To summarize, the program calculates the sums of rows, columns, diagonals, and anti-diagonals of the entered matrix and checks if they all match. If they match, the matrix is considered a magic square.

Write a c program to create a square matrix where only the rows and the column sum is up to the magic constant.(semi magic square)

```
#include <stdio.h>

#define MAX_SIZE 10

void createSemiMagicSquare(int n) {
   int square[MAX_SIZE][MAX_SIZE] = {0};
   int magicConstant = n * (n * n + 1) / 2;
   // Initialize the matrix with 0
   int i, j;
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         square[i][j] = 0;
      }
   }
   int row = 0, col = n / 2, num = 1;
   while (num <= n * n) {
      square[row][col] = num;
      // Move diagonally up and to the right
      row--;
```

```
    col++;
    // If the position goes out of bounds, wrap around to the opposite side
    if (row < 0) {
        row = n - 1;
    }
    if (col == n) {
        col = 0;
    }
    // If the current position is already filled, move vertically down
    if (square[row][col] != 0) {
        row = (row + 1) % n;
        col = (col - 1 + n) % n;
    }
    num++;
}
// Print the semi-magic square
printf("Semi-Magic Square:\n");
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        printf("%2d ", square[i][j]);
    }
    printf("\n");
}
// Verify the sum of each row and column
for (i = 0; i < n; i++) {
    int rowSum = 0, colSum = 0;
    for (j = 0; j < n; j++) {
        rowSum += square[i][j];
        colSum += square[j][i];
    }
    if (rowSum != magicConstant || colSum != magicConstant) {
        printf("Error: Not a semi-magic square!\n");
        return;
    }
}
```

```
    printf("Sum of each row and column: %d\n", magicConstant);
}
int main() {
    int n;
    printf("Enter the size of the semi-magic square: ");
    scanf("%d", &n);

    if (n % 2 == 0) {
        printf("Error: Size should be odd.\n");
        return 0;
    }
    createSemiMagicSquare(n);
    return 0;
}
```

1. The code begins by including the necessary header file `stdio.h`, which provides input and output functions.
2. The constant `MAX_SIZE` is defined to set the maximum size of the square matrix. In this case, it is set to 10, but you can change it according to your needs.
3. The function `createSemiMagicSquare` is defined, which takes an integer `n` as input. This function generates a semi-magic square matrix of size `n`.
4. Inside the `createSemiMagicSquare` function, a 2D array `square` of size `MAX_SIZE` is declared and initialized with zeros. This array will hold the semi-magic square matrix.
5. The variable `magicConstant` is calculated as `n * (n * n + 1) / 2`. In a magic square, the sum of each row, column, and diagonal is equal to the magic constant. Here, we calculate the magic constant based on the size of the square.
6. The nested loops initialize the matrix by setting all elements to zero.
7. The variables `row`, `col`, and `num` are initialized to keep track of the current position in the matrix and the value to be filled in each cell. The starting position is the top-center cell (`row = 0`, `col = n / 2`).
8. The while loop runs until `num` reaches `n * n`, which is the total number of cells in the square matrix.
9. Inside the while loop, the current `num` is assigned to the current position in the matrix `square[row][col]`.
10. The program then moves diagonally up and to the right by decrementing `row` and incrementing `col`.
11. If the new position goes out of bounds, it wraps around to the opposite side of the matrix. This ensures that the semi-magic square is created in a spiral pattern.
12. If the current position is already filled (i.e., not zero), the program moves vertically down by incrementing `row` and decrementing `col`.
13. After filling all the cells in the matrix, the program prints the resulting semi-magic square.
14. The program then verifies the sum of each row and column. It calculates the sum of each row and column and checks if they are equal to the magic constant. If any sum is different, it prints an error message indicating that it is not a semi-magic square.
15. Finally, the magic constant is printed, indicating the sum of each row and column in the semi-magic square.
16. In the `main` function, the user is prompted to enter the size of the semi-magic square.
17. If the size entered is not odd (i.e., `n % 2 == 0`), an error message is displayed, and the program terminates.
18. If the size is odd, the `createSemiMagicSquare` function is called with the entered size `n`.
19. The program exits by returning 0, indicating successful execution.

Write a c program to Check whether the given matrix of order 3x3 is semimagic or not (whose rows and columns add up to a magic number but whose main diagonals do not)

1 5 9

6 7 2

8 3 4

```
#include <stdio.h>

Int isSemimagic(int matrix[3][3]) {
   Int sum1 = 0, sum2 = 0;
   Int I, j;
   // Calculate the sum of the first row
   For (I = 0; I < 3; i++) {
      Sum1 += matrix[0][i];
   }
   // Check whether all rows have the same sum
   For (I = 1; I < 3; i++) {
      Int rowSum = 0;
      For (j = 0; j < 3; j++) {
         rowSum += matrix[i][j];
      }
      If (rowSum != sum1) {
         Return 0; // Not semimagic
      }
   }
   // Check whether all columns have the same sum
   For (j = 0; j < 3; j++) {
      Int colSum = 0;
      For (I = 0; I < 3; i++) {
         colSum += matrix[i][j];
      }
      If (colSum != sum1) {
         Return 0; // Not semimagic
      }
   }
   // Calculate the sum of the main diagonal
   For (I = 0; I < 3; i++) {
      Sum2 += matrix[i][i];
   }
   // Check whether the main diagonal sum is different from the others
   If (sum2 == sum1) {
```

```
    Return 0; // Not semimagic
  }
  // Check whether the secondary diagonal sum is different from the others
  Int sum3 = matrix[0][2] + matrix[1][1] + matrix[2][0];
  If (sum3 == sum1) {
    Return 0; // Not semimagic
  }
  Return 1; // Semimagic
}
Int main() {
  Int matrix[3][3] = {{1, 5, 9},
                      {6, 7, 2},
                      {8, 3, 4}};
  If (isSemimagic(matrix)) {
    Printf("The matrix is semimagic.\n");
  } else {
    Printf("The matrix is not semimagic.\n");
  }
  Return 0;
}
```

Explain

The given code is a C program that checks whether a given matrix of order 3x3 is semimagic or not. A semimagic matrix is one in which the rows and columns add up to a magic number, but the main diagonals do not.

1. The `isSemimagic` function takes a 3x3 matrix as input and returns an integer value indicating whether the matrix is semimagic or not. It uses the following approach to determine semimagicness:
    a. Two integer variables `sum1` and `sum2` are declared to store the sums of the first row and the main diagonal, respectively.
    b. Two nested `for` loops iterate over the matrix to calculate the sum of the first row and check whether all rows have the same sum. If any row has a different sum, the function returns 0, indicating that the matrix is not semimagic.
    c. Another set of nested `for` loops calculates the sum of each column and checks whether all columns have the same sum. If any column has a different sum, the function returns 0.
    d. The sum of the main diagonal is calculated by iterating over the diagonal elements of the matrix.
    e. The function checks whether the sum of the main diagonal is equal to the sum of the first row. If they are equal, the function returns 0, indicating that the matrix is not semimagic.
    f. Finally, the function calculates the sum of the secondary diagonal and checks whether it is equal to the sum of the first row. If they are equal, the function returns 0.
    g. If none of the above conditions are met, the function returns 1, indicating that the matrix is semimagic.
2. In the `main` function, a 3x3 matrix is defined and initialized with the values provided in the example: {{1, 5, 9}, {6, 7, 2}, {8, 3, 4}}.
3. The `isSemimagic` function is called with the matrix as an argument.
4. Depending on the return value of `isSemimagic`, the corresponding message is printed to the console, indicating whether the matrix is semimagic or not.

In this case, since the given matrix satisfies the conditions for a semimagic #

Note that the code assumes a fixed 3x3 matrix size and does not handle input validation or dynamic matrix sizes.

Write a c program to Create and display a square matrix of order 3x3 , where the sum of elements in each row,column and diagonal is 15

```
#include <stdio.h>

#define SIZE 3

Void createMatrix(int matrix[SIZE][SIZE]) {
   Int I, j;
   Int num = 1;
   For (I = 0; I < SIZE; i++) {
      For (j = 0; j < SIZE; j++) {
         Matrix[i][j] = num++;
      }
   }
}
Void displayMatrix(int matrix[SIZE][SIZE]) {
   Int I, j;
   Printf("Square Matrix:\n");
   For (I = 0; I < SIZE; i++) {
      For (j = 0; j < SIZE; j++) {
         Printf("%d\t", matrix[i][j]);
      }
      Printf("\n");
   }
}


Int checkSum(int matrix[SIZE][SIZE]) {
   Int I, j;
   Int sumRow, sumCol, sumDiag1, sumDiag2;
   // Check row sums
   For (I = 0; I < SIZE; i++) {
      sumRow = 0;
      for (j = 0; j < SIZE; j++) {
         sumRow += matrix[i][j];
      }
      If (sumRow != 15)
         Return 0;
```

```
    }
    // Check column sums
    For (j = 0; j < SIZE; j++) {
        sumCol = 0;
        for (I = 0; I < SIZE; i++) {
            sumCol += matrix[i][j];
        }
        If (sumCol != 15)
            Return 0;
    }

    // Check diagonal sums
    sumDiag1 = matrix[0][0] + matrix[1][1] + matrix[2][2];
    sumDiag2 = matrix[0][2] + matrix[1][1] + matrix[2][0];
    if (sumDiag1 != 15 || sumDiag2 != 15)
        return 0;

    return 1;
}

Int main() {
    Int matrix[SIZE][SIZE];

    createMatrix(matrix);
    displayMatrix(matrix);

    if (checkSum(matrix))
        printf("\nThe sum of elements in each row, column, and diagonal is 15.\n");
    else
        printf("\nThe sum of elements in each row, column, and diagonal is not 15.\n");

    return 0;
}
```

Explain

1. The code begins with the inclusion of the standard input/output library `stdio.h`.

2. The constant `SIZE` is defined with a value of 3, representing the order of the square matrix.
3. The `createMatrix` function is defined to populate the matrix with numbers from 1 to 9. It takes a 2D array (`matrix`) as a parameter and uses nested loops to assign consecutive values to each element of the matrix.
4. The `displayMatrix` function is defined to print the matrix on the screen. It also takes the `matrix` array as a parameter and uses nested loops to iterate over each element and print it.
5. The `checkSum` function is defined to validate if the sum of elements in each row, column, and diagonal is equal to 15. It takes the `matrix` array as a parameter and uses additional variables to calculate the sum of rows, columns, and diagonals. It checks the sums of all rows and columns using nested loops, and then separately calculates the sums of the two diagonals. If any sum is not equal to 15, it returns 0 (false). Otherwise, it returns 1 (true).
6. In the `main` function, an `int` matrix of size `SIZE x SIZE` is declared
7. The `createMatrix` function is called, passing the `matrix` array as an argument. This populates the matrix with numbers from 1 to 9.
8. The `displayMatrix` function is called, passing the `matrix` array as an argument. This prints the matrix on the screen.
9. The `checkSum` function is called, passing the `matrix` array as an argument. It checks if the sum of elements in each row, column, and diagonal is 15.
10. Based on the return value of `checkSum`, the program prints a corresponding message indicating whether the sum of elements in each row, column, and diagonal is 15 or not.
11. Finally, the `main` function returns 0 to indicate successful program execution.

The program creates a 3x3 square matrix, populates it with numbers from 1 to 9, and checks if the sum of elements in each row, column, and diagonal is 15. If it is, it displays a success message; otherwise, it displays a failure message.

Mcqs

1. What is the order of a matrix?

A. number of rows multiplied number of columns

B. number of columns multiplied number of rows

C. number of rows multiplied number of rows

D. number of columns multiplied number of columns

2. How do you allocate a matrix using a single pointer in C? (M and N are the number of rows and columns respectively)

A. Int *arr = malloc(M * N * sizeof(int));

B. int *arr = (int *)malloc(M * N * sizeof(int));

C. int *arr = (int *)malloc(M + N * sizeof(int));

D. int *arr = (int *)malloc(M * N * sizeof(arr));

3.Which of the following don’t use matrices?

A. In solving linear equations

B. Image processing

C. Graph theory

D. Sorting numbers

4. Matrix A when multiplied with Matrix C gives the Identity matrix I, what is C?

A. Identity matrix

B. Inverse of A

C. Square of A

D. Transpose of A

5. What is the relation between Sparsity and Density of a matrix?

A. Sparsity = 1 – Density

B. Sparsity = 1 + Density

C. Sparsity = Density*Total number of elements

D. Sparsity = Density/Total number of elements

6. Who coined the term Matrix?

A. Harry Markowitz

B. James Sylvester

C. Chris Messina

D. Arthur Cayley

7. Who coined the term Sparse Matrix?

A. Harry Markowitz

B. James Sylvester

C. Chris Messina

D. Arthur Cayley

8.Which of the following is not the method to represent Sparse Matrix?

A. Dictionary of Keys

B. Linked List

C. Array

D. Heap

9. Which one of the following is a Special Sparse Matrix?

A. Band Matrix

B. Skew Matrix

C. Null matrix

D. Unit matrix

10. Which of the following is the way to represent Sparse Matrix?

A. Arra

B. Linked list

C. Both A and B

D. None of the above

1. A memory for sparse matrix is dedicated by the _____ command. (a) spalloc (b) sparsealloc (c) allocspar (d) no such command 2. The default number of non-zero elements which can be put into the memory allocated by the spalloc command is > 1. (a) True (b) False 3. A power pattern for an antenna is a _____ (a) 4-D plot of space and power (b) 3-D plot of space and power (c) 2-D plot of space and power (d) Nothing called power pattern 4. The if structure is a _____ (a) Conditional structure (b) Logical structure (c) Nested structure (d) Biased structure 5. What is a sparse array?

A.) Data structure for representing arrays of records

B.) Data structure that compactly stores bits

C.) An array in which most of the elements have the same value

D.) None of these

Write a c program for Addition of two Single Variable Polynomials, using the representation where the exponents are implicit

```
#include <stdio.h>

#include <stdlib.h>


struct Node {

   int coefficient;

   struct Node* next;

};


struct Node* createNode(int coefficient) {

   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

   newNode->coefficient = coefficient;

   newNode->next = NULL;

   return newNode;

}


struct Node* addPolynomials(struct Node* poly1, struct Node* poly2) {

   struct Node* result = NULL;

   struct Node* temp, *prev = NULL;

   int sum;


   while (poly1 != NULL && poly2 != NULL) {

      if (poly1->coefficient == 0) {

         poly1 = poly1->next;

         continue;
```

```
    }
    if (poly2->coefficient == 0) {
        poly2 = poly2->next;
        continue;
    }

    if (poly1->coefficient > poly2->coefficient) {
        sum = poly1->coefficient;
        poly1 = poly1->next;
    } else if (poly1->coefficient < poly2->coefficient) {
        sum = poly2->coefficient;
        poly2 = poly2->next;
    } else {
        sum = poly1->coefficient + poly2->coefficient;
        poly1 = poly1->next;
        poly2 = poly2->next;
    }

    temp = createNode(sum);

    if (result == NULL)
        result = temp;
    else
        prev->next = temp;

    prev = temp;
}

while (poly1 != NULL) {
    if (poly1->coefficient != 0) {
        temp = createNode(poly1->coefficient);
        prev->next = temp;
        prev = temp;
    }
    poly1 = poly1->next;
```

```
    }

    while (poly2 != NULL) {
        if (poly2->coefficient != 0) {
            temp = createNode(poly2->coefficient);
            prev->next = temp;
            prev = temp;
        }
        poly2 = poly2->next;
    }

    return result;
}

void displayPolynomial(struct Node* poly) {
    struct Node* temp = poly;
    if (temp == NULL) {
        printf("0\n");
        return;
    }
    while (temp != NULL) {
        if (temp->coefficient != 0) {
            printf("%dx^%d", temp->coefficient, temp->coefficient);
            if (temp->next != NULL && temp->next->coefficient != 0) {
                printf(" + ");
            }
        }
        temp = temp->next;
    }
    printf("\n");
}

void freePolynomial(struct Node* poly) {
    struct Node* temp;
    while (poly != NULL) {
```

```
        temp = poly;
        poly = poly->next;
        free(temp);
    }
}

int main() {
    struct Node* poly1 = NULL;
    struct Node* poly2 = NULL;
    struct Node* result = NULL;
    int coeff, degree, terms, i;

    printf("Enter the number of terms in polynomial 1: ");
    scanf("%d", &terms);

    printf("Enter the coefficient and degree for each term in polynomial 1:\n");
    for (i = 0; i < terms; i++) {
        scanf("%d %d", &coeff, &degree);
        struct Node* newNode = createNode(coeff);
        newNode->coefficient = degree;
        newNode->next = poly1;
        poly1 = newNode;
    }

    printf("Enter the number of terms in polynomial 2: ");
    scanf("%d", &terms);

    printf("Enter the coefficient and degree for each term in polynomial 2:\n");
    for (i = 0; i < terms; i++) {
        scanf("%d %d", &coeff, &degree);
        struct Node* newNode = createNode(coeff);
        newNode->coefficient = degree;
        newNode->next = poly2;
        poly2 = newNode;
    }
```

```
    printf("\nPolynomial 1: ");
    displayPolynomial(poly1);
    printf("Polynomial 2: ");
    displayPolynomial(poly2);
    result = addPolynomials(poly1, poly2);
    printf("\nSum of the polynomials: ");
    displayPolynomial(result);
    freePolynomial(poly1);
    freePolynomial(poly2);
    freePolynomial(result);
    return 0;
}
```

1. The code defines a structure `Node` that represents a single term in the polynomial. It has two fields: `coefficient` to store the coefficient of the term and `next` to point to the next term in the polynomial.
2. The function `createNode` creates a new node with the given coefficient and returns a pointer to it.
3. The function `addPolynomials` takes two polynomial linked lists (`poly1` and `poly2`) and adds them together to produce a new polynomial linked list (`result`). It iterates through the two lists simultaneously, considering the coefficients of the terms. If the coefficients are equal, it adds them together. If one coefficient is greater than the other, it adds that coefficient alone. The resulting terms are appended to the `result` list.
4. The function `displayPolynomial` takes a polynomial linked list and prints it in the form of a polynomial expression. It iterates through the list and prints each term's coefficient and degree. It skips terms with a coefficient of 0.
5. The function `freePolynomial` frees the memory allocated for a polynomial linked list by iterating through the list and deallocating each node.
6. In the `main` function, the user is prompted to enter the number of terms and the coefficient with the corresponding degree for each term in two polynomials (`poly1` and `poly2`). The input values are used to create the polynomial linked lists using the `createNode` function.
7. The polynomials `poly1` and `poly2` are displayed using the `displayPolynomial` function.
8. The `addPolynomials` function is called to add the two polynomials together, and the result is stored in the `result` variable.
9. The sum of the polynomials is displayed using the `displayPolynomial` function.
10. The memory allocated for the polynomial linked lists (`poly1`, `poly2`, and `result`) is freed using the `freePolynomial` function.

Write a c program for Multiplication of two Single Variable Polynomials, using the representation where the exponents are implict

```
#include <stdio.h>
#define MAX_TERMS 100
typedef struct {
    int coefficient;
} Term;
void multiplyPolynomials(Term polynomial1[], int terms1, Term polynomial2[], int terms2, Term result[]) {
    int i, j, k;
    // Initialize the result polynomial
    for (i = 0; i < MAX_TERMS; i++) {
        result[i].coefficient = 0;
    }
```

```
    // Multiply the polynomials
    for (i = 0; i < terms1; i++) {
        for (j = 0; j < terms2; j++) {
            k = i + j;
            result[k].coefficient += polynomial1[i].coefficient * polynomial2[j].coefficient;
        }
    }
}
void displayPolynomial(Term polynomial[], int terms) {
    int i;
    for (i = 0; i < terms; i++) {
        printf("%d ", polynomial[i].coefficient);
    }
    printf("\n");
}
int main() {
    Term polynomial1[MAX_TERMS], polynomial2[MAX_TERMS], result[MAX_TERMS];
    int terms1, terms2;
    printf("Enter the number of terms in polynomial 1: ");
    scanf("%d", &terms1);
    printf("Enter the coefficients of polynomial 1: ");
    for (int i = 0; i < terms1; i++) {
        scanf("%d", &polynomial1[i].coefficient);
    }
    printf("Enter the number of terms in polynomial 2: ");
    scanf("%d", &terms2);
    printf("Enter the coefficients of polynomial 2: ");
    for (int i = 0; i < terms2; i++) {
        scanf("%d", &polynomial2[i].coefficient);
    }
    multiplyPolynomials(polynomial1, terms1, polynomial2, terms2, result);
    printf("Resultant Polynomial: ");
    displayPolynomial(result, terms1 + terms2 - 1);
    return 0;
}
```

Explain

The program starts by defining a structure `Term` which represents a single term of a polynomial. Each term consists of an integer coefficient.

The function `multiplyPolynomials` takes two input polynomials `polynomial1` and `polynomial2`, along with their respective term counts `terms1` and `terms2`. It also takes an output parameter `result` to store the product of the two polynomials.

Inside the `multiplyPolynomials` function, the result polynomial is initialized by setting all coefficients to zero.

The function then performs the multiplication of the polynomials using nested loops. It iterates over each term in `polynomial1` and `polynomial2`, multiplies their coefficients, and accumulates the results in the corresponding terms of the `result` polynomial.

After multiplying the polynomials, the `displayPolynomial` function is called to print the coefficients of the resulting polynomial.

In the `main` function, the user is prompted to enter the number of terms and coefficients for both polynomials. The input coefficients are stored in the respective polynomial arrays.

The `multiplyPolynomials` function is called with the input polynomials, and the resulting polynomial is stored in the `result` array.

Finally, the program displays the resulting polynomial by calling the `displayPolynomial`

execution:Example Run:

Enter the number of terms in polynomial 1: 3

Enter the coefficients of polynomial 1: 2 4 1

Enter the number of terms in polynomial 2: 2

Enter the coefficients of polynomial 2: 3 2

Resultant Polynomial: 6 14 11 2

In this example, we have two polynomials:1Polynomial 1: 2x^2 + 4x 1...Polynomial 2: 3x + 2 The program multiplies these polynomials and displays the resulting polynomial: 6x^3 + 14x^2 + 11x + 2.

Write a c program for Evaluation of Single Variable Polynomial, using the representation where the exponents are implicit

```
#include <stdio.h>


// Function to evaluate a single-variable polynomial
Double evaluatePolynomial(double coefficients[], int degree, double x) {
   Double result = 0;
   Int I;

   // Evaluate the polynomial using Horner's method
   For (I = degree; I >= 0; i--) {
      Result = result * x + coefficients[i];
   }

   Return result;
}

Int main() {
```

```
  Int degree, I;

  Printf(“Enter the degree of the polynomial: “);
  Scanf(“%d”, &degree);

  // Array to store the coefficients
  Double coefficients[degree + 1];

  Printf(“Enter the coefficients in descending order:\n”);
  For (I = degree; I >= 0; i--) {
    Printf(“Coefficient of x^%d: “, i);
    Scanf(“%lf”, &coefficients[i]);
  }

  Double x;

  Printf(“Enter the value of x: “);
  Scanf(“%lf”, &x);

  // Evaluate the polynomial
  Double result = evaluatePolynomial(coefficients, degree, x);

  Printf(“The result of evaluating the polynomial at x = %.2lf is %.2lf\n”, x, result);

  Return 0;
}
```

Explain

1. The code begins by including the necessary header file `stdio.h`, which provides input/output functions like `printf` and `scanf`.
2. Next, there is a function `evaluatePolynomial` that takes three parameters: `coefficients[]`, `degree`, and `x`. It calculates and returns the value of the polynomial for the given `x` value using Horner’s method. The function initializes `result` to 0 and iterates over the coefficients in reverse order, multiplying the accumulated result by `x` and adding the current coefficient. Finally, it returns the computed result.
3. The `main` function is where the program execution starts. It declares variables `degree` and `i`.
4. The user is prompted to enter the degree of the polynomial using `printf` and `scanf`. The input is stored in the variable `degree`.
5. An array `coefficients[]` is declared with a size of `degree + 1`. This array will store the coefficients of the polynomial.
6. The user is prompted to enter the coefficients in descending order. Inside a loop, starting from the highest degree down to 0, the program asks for the coefficient of each term and stores it in the corresponding index of the `coefficients[]` array.
7. The user is then prompted to enter the value of `x` at which the polynomial needs to be evaluated. The input is stored in the variable `x`.
8. The `evaluatePolynomial` function is called with the `coefficients[]` array, `degree`, and `x` as arguments, and the result is stored in the variable `result`.

9. Finally, the program prints the result of evaluating the polynomial at the given `x` value using `printf`. The format specifier `%.2lf` ensures that the output is displayed with two decimal places.
10. The `return 0` statement indicates that the program has executed successfully.

This program allows you to enter the degree and coefficients of a single-variable polynomial, as well as the value of `x`, and it calculates the result by evaluating the polynomial at `x` using Horner's method.

Write a c program for Addition of two Multivariable Polynomials with two variables, using the representation where the exponents are implicit

```
#include <stdio.h>


// Structure to represent a term in the polynomial

Typedef struct

{

   Int coeff;

   Int var1;

   Int var2;

} Term;


// Function to add two multivariable polynomials

Void addPolynomials(Term poly1[], int size1, Term poly2[], int size2, Term result[], int *sizeResult)

{

   Int I = 0, j = 0, k = 0;


   While (I < size1 && j < size2)

   {

      If (poly1[i].var1 > poly2[j].var1 || (poly1[i].var1 == poly2[j].var1 && poly1[i].var2 > poly2[j].var2))

      {

         Result[k++] = poly1[i++];

      }

      Else if (poly1[i].var1 < poly2[j].var1 || (poly1[i].var1 == poly2[j].var1 && poly1[i].var2 < poly2[j].var2))

      {

         Result[k++] = poly2[j++];

      }

      Else

      {

         Result[k].var1 = poly1[i].var1;

         Result[k].var2 = poly1[i].var2;

         Result[k].coeff = poly1[i].coeff + poly2[j].coeff;
```

```
            I++;
            J++;
            K++;
        }
    }

    While (I < size1)
    {
        Result[k++] = poly1[i++];
    }

    While (j < size2)
    {
        Result[k++] = poly2[j++];
    }

    *sizeResult = k;
}

// Function to display a polynomial
Void displayPolynomial(Term poly[], int size)
{
    Int I;

    For (I = 0; I < size; i++)
    {
        Printf("%dx^%dy^%d ", poly[i].coeff, poly[i].var1, poly[i].var2);
        If (I < size – 1)
        {
            Printf("+ ");
        }
    }

    Printf("\n");
}
```

```
Int main()
{
    Term poly1[100], poly2[100], result[200];
    Int size1, size2, sizeResult;

    Printf("Enter the number of terms in the first polynomial: ");
    Scanf("%d", &size1);

    Printf("Enter the coefficients and variables (var1 var2 coeff) of the first polynomial:\n");
    For (int I = 0; I < size1; i++)
    {
        Scanf("%d %d %d", &poly1[i].var1, &poly1[i].var2, &poly1[i].coeff);
    }

    Printf("Enter the number of terms in the second polynomial: ");
    Scanf("%d", &size2);

    Printf("Enter the coefficients and variables (var1 var2 coeff) of the second polynomial:\n");
    For (int I = 0; I < size2; i++)
    {
        Scanf("%d %d %d", &poly2[i].var1, &poly2[i].var2, &poly2[i].coeff);
    }

    addPolynomials(poly1, size1, poly2, size2, result, &sizeResult);

    printf("Resultant polynomial after addition: ");
    displayPolynomial(result, sizeResult);

    return 0;
}
```

1. We start by defining a structure called `Term` to represent a term in the polynomial. It consists of three fields: `coeff` (coefficient), `var1` (exponent of the first variable), and `var2` (exponent of the second variable).
2. The `addPolynomials` function takes two polynomial arrays (`poly1` and `poly2`) along with their respective sizes (`size1` and `size2`), an array to store the result (`result`), and a pointer to an integer (`sizeResult`) to store the size of the resulting polynomial.
3. Inside the `addPolynomials` function, we use three variables `i`, `j`, and `k` to iterate through the arrays `poly1`, `poly2`, and `result` respectively.
4. The while loop compares the exponents (`var1` and `var2`) of the terms in `poly1` and `poly2` and adds them to `result` based on the comparison. If the exponents are equal, we add the coefficients and store the result in `result`. If the exponents are not equal, we copy the term with the higher exponent to `result`.
5. After adding all the terms from `poly1` and `poly2` to `result`, we copy any remaining terms from either polynomial to `result`
6. Finally, we update the value of `sizeResult` with the number of terms in the resulting polynomial.

7. The `displayPolynomial` function takes a polynomial array (`poly`) and its size (`size`) as input and displays the polynomial in a readable format.
8. In the `main` function, we declare arrays `poly1`, `poly2`, and `result` to store the polynomials. We also declare variables `size1`, `size2`, and `sizeResult` to store the sizes of the polynomials.
9. We prompt the user to enter the number of terms in the first polynomial and read the input using `scanf`. Then, we prompt the user to enter the coefficients and variables (exponents) of each term in the first polynomial and store them in `poly1`.
10. Similarly, we ask the user for the number of terms in the second polynomial, read the input, and store the coefficients and variables in `poly2`.
11. We call the `addPolynomials` function, passing `poly1`, `size1`, `poly2`, `size2`, `result`, and `sizeResult` as arguments.
12. Finally, we display the resulting polynomial by calling the `displayPolynomial` function with `result` and `sizeResult` as arguments.

That's it! The program takes two multivariable polynomials as input, adds them together, and displays the resulting polynomial.

Write a c program for Multiplication of two Multivariable Polynomials with two variables, using the representation where the exponents are implicit

```
#include <stdio.h>


// Structure to represent a term in the polynomial

Typedef struct

{

  Int coeff;

  Int var1;

  Int var2;

} Term;


// Function to multiply two multivariable polynomials

Void multiplyPolynomials(Term poly1[], int size1, Term poly2[], int size2, Term result[], int *sizeResult)

{

  Int I, j, k = 0;


  For (I = 0; I < size1; i++)

  {

    For (j = 0; j < size2; j++)

    {

      Result[k].coeff = poly1[i].coeff * poly2[j].coeff;

      Result[k].var1 = poly1[i].var1 + poly2[j].var1;

      Result[k].var2 = poly1[i].var2 + poly2[j].var2;

      K++;

    }

  }
```

```
    *sizeResult = k;
}

// Function to display a polynomial
Void displayPolynomial(Term poly[], int size)
{
    Int I;

    For (I = 0; I < size; i++)
    {
        Printf("%dx^%dy^%d ", poly[i].coeff, poly[i].var1, poly[i].var2);
        If (I < size – 1)
        {
            Printf("+ ");
        }
    }

    Printf("\n");
}

Int main()
{
    Term poly1[100], poly2[100], result[10000];
    Int size1, size2, sizeResult;

    Printf("Enter the number of terms in the first polynomial: ");
    Scanf("%d", &size1);

    Printf("Enter the coefficients and variables (var1 var2 coeff) of the first polynomial:\n");
    For (int I = 0; I < size1; i++)
    {
        Scanf("%d %d %d", &poly1[i].var1, &poly1[i].var2, &poly1[i].coeff);
    }
```

```
    Printf("Enter the number of terms in the second polynomial: ");
    Scanf("%d", &size2);

    Printf("Enter the coefficients and variables (var1 var2 coeff) of the second polynomial:\n");
    For (int I = 0; I < size2; i++)
    {
        Scanf("%d %d %d", &poly2[i].var1, &poly2[i].var2, &poly2[i].coeff);
    }

    multiplyPolynomials(poly1, size1, poly2, size2, result, &sizeResult);

    printf("Resultant polynomial after multiplication: ");
    displayPolynomial(result, sizeResult);

    return 0;
}
```

Explain

Certainly! Here's a C program that performs multiplication of two multivariable polynomials with two variables, assuming the representation where the exponents are implicit:

```c
#include <stdio.h>

// Structure to represent a term in the polynomial
Typedef struct
{
    Int coeff;
    Int var1;
    Int var2;
} Term;

// Function to multiply two multivariable polynomials
Void multiplyPolynomials(Term poly1[], int size1, Term poly2[], int size2, Term result[], int *sizeResult)
{
    Int I, j, k = 0;
```

```
   For (I = 0; I < size1; i++)
   {
      For (j = 0; j < size2; j++)
      {
         Result[k].coeff = poly1[i].coeff * poly2[j].coeff;
         Result[k].var1 = poly1[i].var1 + poly2[j].var1;
         Result[k].var2 = poly1[i].var2 + poly2[j].var2;
         K++;
      }
   }

   *sizeResult = k;
}

// Function to display a polynomial
Void displayPolynomial(Term poly[], int size)
{
   Int I;

   For (I = 0; I < size; i++)
   {
      Printf("%dx^%dy^%d ", poly[i].coeff, poly[i].var1, poly[i].var2);
      If (I < size – 1)
      {
         Printf("+ ");
      }
   }

   Printf("\n");
}

Int main()
{
   Term poly1[100], poly2[100], result[10000];
```

```
  Int size1, size2, sizeResult;

  Printf("Enter the number of terms in the first polynomial: ");
  Scanf("%d", &size1);

  Printf("Enter the coefficients and variables (var1 var2 coeff) of the first polynomial:\n");
  For (int I = 0; I < size1; i++)
  {
    Scanf("%d %d %d", &poly1[i].var1, &poly1[i].var2, &poly1[i].coeff);
  }

  Printf("Enter the number of terms in the second polynomial: ");
  Scanf("%d", &size2);

  Printf("Enter the coefficients and variables (var1 var2 coeff) of the second polynomial:\n");
  For (int I = 0; I < size2; i++)
  {
    Scanf("%d %d %d", &poly2[i].var1, &poly2[i].var2, &poly2[i].coeff);
  }

  multiplyPolynomials(poly1, size1, poly2, size2, result, &sizeResult);

  printf("Resultant polynomial after multiplication: ");
  displayPolynomial(result, sizeResult);

  return 0;
}
```

Explain

1. We start by defining a structure called `Term` to represent a term in the polynomial, which includes fields for coefficient (`coeff`), variable 1 (`var1`), and variable 2 (`var2`).
2. The `multiplyPolynomials` function takes two polynomial arrays (`poly1` and `poly2`) along with their respective sizes (`size1` and `size2`), an array to store the result (`result`), and a pointer to an integer (`sizeResult`) to store the size of the resulting polynomial.
3. Inside the `multiplyPolynomials` function, we use two nested loops to iterate through each term in `poly1` and `poly2`. For each pair of terms, we perform the multiplication and store tht

Result in `result`. The multiplication is done by multiplying the coefficients and adding the exponents of the variables.

4. After multiplying all the terms, we update the value of `sizeResult` with the number of terms in the resulting polynomial.
5. The `displayPolynomial` function takes a polynomial array (`poly`) and its size (`size`) as input and displays the polynomial in a readable format.

6. In the `main` function, we declare arrays `poly1`, `poly2`, and `result` to store the polynomials. We also declare variables `size1`, `size2`, and `sizeResult` to store the sizes of the polynomials.
7. We prompt the user to enter the number of terms in the first polynomial and read the input using `scanf`. Then, we prompt the user to enter the coefficients and variables (exponents) of each term in the first polynomial and store them in `poly1`.
8. Similarly, we ask the user for the number of terms in the second polynomial, read the input, and store the coefficients and variables in `poly2`.
9. We call the `multiplyPolynomials` function, passing `poly1`, `size1`, `poly2`, `size2`, `result`, and `sizeResult` as arguments.
10. Finally, we display the resulting polynomial by calling the `displayPolynomial` function with `result` and `sizeResult` as arguments.

That's it! The program takes two multivariable polynomials as input, performs multiplication on them, and displays the resulting polynomial.

Write a c program for Addition of two Single Variable Polynomials, using representation where the exponents are explicitly specified

```
#include <stdio.h>

#include <stdlib.h>


Struct Term {

  Float coefficient;

  Int exponent;

};


Struct Polynomial {

  Struct Term *terms;

  Int degree;

};


Void addPolynomials(struct Polynomial p1, struct Polynomial p2, struct Polynomial *result) {

  Int I = 0, j = 0, k = 0;


  While (I < p1.degree && j < p2.degree) {

    If (p1.terms[i].exponent > p2.terms[j].exponent) {

      Result->terms[k++] = p1.terms[i++];

    } else if (p1.terms[i].exponent < p2.terms[j].exponent) {

      Result->terms[k++] = p2.terms[j++];

    } else {

      Float sum = p1.terms[i].coefficient + p2.terms[j].coefficient;

      If (sum != 0) {

        Result->terms[k].coefficient = sum;

        Result->terms[k++].exponent = p1.terms[i].exponent;

      }

      I++;
```

```
        J++;
      }
    }

    While (I < p1.degree) {
      Result->terms[k++] = p1.terms[i++];
    }

    While (j < p2.degree) {
      Result->terms[k++] = p2.terms[j++];
    }

    Result->degree = k;
}

Void displayPolynomial(struct Polynomial p) {
    For (int I = 0; I < p.degree; i++) {
      If (p.terms[i].coefficient != 0) {
        Printf("%.2fx^%d ", p.terms[i].coefficient, p.terms[i].exponent);
        If (I < p.degree – 1) {
          Printf("+ ");
        }
      }
    }
    Printf("\n");
}

Int main() {
    Struct Polynomial p1, p2, sum;
    Int degree1, degree2;

    Printf("Enter the degree of polynomial 1: ");
    Scanf("%d", &degree1);
    P1.degree = degree1 + 1;
    P1.terms = (struct Term *)malloc(p1.degree * sizeof(struct Term));
```

```
Printf(“Enter the coefficients for each term in polynomial 1:\n”);
For (int I = 0; I < p1.degree; i++) {
   Printf(“Coefficient of x^%d: “, i);
   Scanf(“%f”, &(p1.terms[i].coefficient));
   P1.terms[i].exponent = I;
}

Printf(“Enter the degree of polynomial 2: “);
Scanf(“%d”, &degree2);
P2.degree = degree2 + 1;
P2.terms = (struct Term *)malloc(p2.degree * sizeof(struct Term));

Printf(“Enter the coefficients for each term in polynomial 2:\n”);
For (int I = 0; I < p2.degree; i++) {
   Printf(“Coefficient of x^%d: “, i);
   Scanf(“%f”, &(p2.terms[i].coefficient));
   P2.terms[i].exponent = I;
}

Sum.terms = (struct Term *)malloc((p1.degree + p2.degree) * sizeof(struct Term));
addPolynomials(p1, p2, &sum);

printf(“\nPolynomial 1: “);
displayPolynomial(p1);

printf(“Polynomial 2: “);
displayPolynomial(p2);

printf(“Sum: “);
displayPolynomial(sum);

free(p1.terms);
free(p2.terms);
free(sum.terms);
```

```
    return 0;
}
```

Explain

This C program allows the user to input two polynomials represented by their degree and coefficients. The polynomials are represented using the `Polynomial` structure, which contains an array of `Term` structures. Each `Term` structure represents a term in the polynomial with its coefficient and exponent.

The `addPolynomials` function takes two `Polynomial` structures `p1` and `p2` as input, along with a pointer to a `Polynomial` structure `result`. It performs the addition of the two polynomials by iterating over their terms. If the exponents of the terms are equal, it adds their coefficients and stores the sum in the `result` polynomial. If the exponents are different, it simply copies the term from the polynomial with the higher exponent into the `result` polynomial. After iterating over all the terms, the function updates the `degree` of the `result` polynomial based on the number of terms in it.

The `displayPolynomial` function takes a `Polynomial` structure as input and prints the polynomial in a readable format. It iterates over the terms of the polynomial, printing the coefficient and exponent for each term. It also adds a "+" sign between terms, except for the last term.

In the `main` function, the user is prompted to enter the degree and coefficients for each term of the two polynomials. The input is stored in the corresponding `Polynomial` structures. Memory is dynamically allocated to store the terms of the polynomials using the `malloc` function.

Then, the `addPolynomials` function is called with the two input polynomials (`p1` and `p2`) and a `sum` polynomial to store the result. The `displayPolynomial` function is used to print the input polynomials and their sum.

Finally, the dynamically allocated memory for the terms of the polynomials is freed using the `free` function to avoid memory leaks.

Write a c program for Multiplication of two Single Variable Polynomials, using representation where the exponents are explicitly specified

```
#include <stdio.h>

#define MAX_DEGREE 100

typedef struct {
    int degree;
    int coefficients[MAX_DEGREE + 1];
} Polynomial;

void initializePolynomial(Polynomial *poly) {
    int i;
    poly->degree = 0;
    for (i = 0; i <= MAX_DEGREE; i++) {
        poly->coefficients[i] = 0;
    }
}

void readPolynomial(Polynomial *poly) {
    int i;
```

```
    printf("Enter the degree of the polynomial: ");
    scanf("%d", &poly->degree);
    printf("Enter the coefficients starting from the lowest degree term:\n");
    for (i = 0; i <= poly->degree; i++) {
        printf("Coefficient of x^%d: ", i);
        scanf("%d", &poly->coefficients[i]);
    }
}

void multiplyPolynomials(const Polynomial *poly1, const Polynomial *poly2, Polynomial *result) {
    int i, j;

    result->degree = poly1->degree + poly2->degree;

    for (i = 0; i <= poly1->degree; i++) {
        for (j = 0; j <= poly2->degree; j++) {
            result->coefficients[i + j] += poly1->coefficients[i] * poly2->coefficients[j];
        }
    }
}

void printPolynomial(const Polynomial *poly) {
    int i;
    printf("Result: ");
    for (i = 0; i <= poly->degree; i++) {
        printf("%dx^%d ", poly->coefficients[i], i);
        if (i < poly->degree) {
            printf("+ ");
        }
    }
    printf("\n");
}

int main() {
    Polynomial poly1, poly2, result;
```

```
    printf("Enter details of the first polynomial:\n");
    initializePolynomial(&poly1);
    readPolynomial(&poly1);

    printf("Enter details of the second polynomial:\n");
    initializePolynomial(&poly2);
    readPolynomial(&poly2);

    multiplyPolynomials(&poly1, &poly2, &result);

    printPolynomial(&result);

    return 0;
}
```

This program performs multiplication of two single variable polynomials using the representation where the exponents are explicitly specified. Here's how the program works:

1. The program defines a struct called `Polynomial`, which represents a polynomial. It consists of two members: `degree`, which stores the degree of the polynomial, and `coefficients`, which is an array that holds the coefficients of each term in the polynomial.

2. The program also defines several functions:

- `initializePolynomial`: This function initializes a polynomial by setting its degree to 0 and initializing all coefficients to 0.

- `readPolynomial`: This function prompts the user to enter the degree and coefficients of a polynomial, and stores the values in the corresponding members of the polynomial struct.

- `multiplyPolynomials`: This function takes two input polynomials (`poly1` and `poly2`) and multiplies them together, storing the result in the `result` polynomial. The function uses nested loops to perform the multiplication, iterating over each term in both polynomials and multiplying the corresponding coefficients.

- `printPolynomial`: This function prints the coefficients and exponents of a polynomial in a readable format.

2. In the `main` function, the program creates three instances of the `Polynomial` struct: `poly1`, `poly2`, and `result`.
3. The program prompts the user to enter the details of the first polynomial by calling `initializePolynomial` and `readPolynomial` functions for `poly1`.
4. Similarly, the program prompts the user to enter the details of the second polynomial by calling the same functions for `poly2`.
5. The program then calls the `multiplyPolynomials` function, passing in `poly1`, `poly2`, and `result`. This function performs the multiplication of the two polynomials and stores the result in the `result` polynomial.
6. Finally, the program calls the `printPolynomial` function to display the result of the multiplication.

This program allows you to input the degree and coefficients of two single variable polynomials and calculates their product, providing the result as a polynomial with the coefficients and corresponding exponents.

Write a small c program for Evaluation of Single Variable Polynomial, using representation where the exponents are explicitly specified

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

```
#define MAX_DEGREE 100

Typedef struct {
   Int degree;
   Int coefficients[MAX_DEGREE + 1];
} Polynomial;

Void initializePolynomial(Polynomial *poly) {
   Int I;
   Poly->degree = 0;
   For (I = 0; I <= MAX_DEGREE; i++) {
      Poly->coefficients[i] = 0;
   }
}

Void readPolynomial(Polynomial *poly) {
   Int I;
   Printf("Enter the degree of the polynomial: ");
   Scanf("%d", &poly->degree);
   Printf("Enter the coefficients starting from the lowest degree term:\n");
   For (I = 0; I <= poly->degree; i++) {
      Printf("Coefficient of x^%d: ", i);
      Scanf("%d", &poly->coefficients[i]);
   }
}

Int evaluatePolynomial(const Polynomial *poly, int x) {
   Int I;
   Int result = 0;
   Int term = 1;

   For (I = 0; I <= poly->degree; i++) {
      Result += poly->coefficients[i] * term;
      Term *= x;
```

```
    }

    Return result;
}

Int main() {
    Polynomial poly;
    Int x, result;

    Printf("Enter details of the polynomial:\n");
    initializePolynomial(&poly);
    readPolynomial(&poly);

    printf("Enter the value of x: ");
    scanf("%d", &x);

    result = evaluatePolynomial(&poly, x);

    printf("Result: %d\n", result);

    return 0;
}
```

Explain

Certainly! Here's a C program that evaluates a single variable polynomial using the representation where the exponents are explicitly specified:

```c
#include <stdio.h>

#define MAX_DEGREE 100

Typedef struct {
    Int degree;
    Int coefficients[MAX_DEGREE + 1];
} Polynomial;
```

```
Void initializePolynomial(Polynomial *poly) {
   Int I;
   Poly->degree = 0;
   For (I = 0; I <= MAX_DEGREE; i++) {
      Poly->coefficients[i] = 0;
   }
}

Void readPolynomial(Polynomial *poly) {
   Int I;
   Printf("Enter the degree of the polynomial: ");
   Scanf("%d", &poly->degree);
   Printf("Enter the coefficients starting from the lowest degree term:\n");
   For (I = 0; I <= poly->degree; i++) {
      Printf("Coefficient of x^%d: ", i);
      Scanf("%d", &poly->coefficients[i]);
   }
}

Int evaluatePolynomial(const Polynomial *poly, int x) {
   Int I;
   Int result = 0;
   Int term = 1;

   For (I = 0; I <= poly->degree; i++) {
      Result += poly->coefficients[i] * term;
      Term *= x;
   }

   Return result;
}

Int main() {
   Polynomial poly;
   Int x, result;
```

```
    Printf("Enter details of the polynomial:\n");

    initializePolynomial(&poly);

    readPolynomial(&poly);


    printf("Enter the value of x: ");

    scanf("%d", &x);


    result = evaluatePolynomial(&poly, x);


    printf("Result: %d\n", result);


    return 0

}
```

1. The program defines a struct called `Polynomial` to represent a polynomial. It has two members: `degree`, which stores the degree of the polynomial, and `coefficients`, an array that holds the coefficients of each term.

2. The program provides three functions:

- `initializePolynomial`: This function initializes a polynomial by setting its degree to 0 and initializing all coefficients to 0.

- `readPolynomial`: This function prompts the user to enter the degree and coefficients of the polynomial, and stores the values in the corresponding members of the `Polynomial` struct.

- `evaluatePolynomial`: This function takes a polynomial (`poly`) and a value of `x` as input. It evaluates the polynomial for the given `x` value using the Horner's method, where each term is multiplied by `x` and accumulated. The result is returned.

2. In the `main` function, the program creates an instance of the `Polynomial` struct called `poly`, and variables `x` and `result`.
3. The program prompts the user to enter the details of the polynomial by calling `initializePolynomial` and `readPolynomial` functions for `poly`.
4. The program then prompts the user to enter a value for `x`.
5. The program calls the `evaluatePolynomial` function, passing in the polynomial `poly` and the value of `x`. The function evaluates the polynomial for the given `x` value and returns the result, which is stored in the variable `result`.
6. Finally, the program prints the result of the polynomial evaluation.

This program allows you to input the degree and coefficients of a single variable polynomial and evaluate it for a specific value of `x`, providing the result as an integer.

Write a small c program for Addition of two Multivariable Polynomials with two variables, using representation where the exponents are explicitly specified

```
#include <stdio.h>


// Structure to represent a term in a polynomial

Struct Term {
```

```
    Int coeff;
    Int expX;
    Int expY;
};

// Function to add two polynomials
Void addPolynomials(struct Term p1[], int size1, struct Term p2[], int size2, struct Term result[]) {
    Int I = 0, j = 0, k = 0;

    // Iterate until any of the polynomials reaches its end
    While (I < size1 && j < size2) {
        // If p1 term has higher total exponent, add it to the result
        If ((p1[i].expX + p1[i].expY) > (p2[j].expX + p2[j].expY)) {
            Result[k++] = p1[i++];
        }
        // If p2 term has higher total exponent, add it to the result
        Else if ((p1[i].expX + p1[i].expY) < (p2[j].expX + p2[j].expY)) {
            Result[k++] = p2[j++];
        }
        // If both terms have the same total exponent, add their coefficients and exponents
        Else {
            Result[k].coeff = p1[i].coeff + p2[j].coeff;
            Result[k].expX = p1[i].expX;
            Result[k].expY = p1[i].expY;
            I++;
            J++;
            K++;
        }
    }

    // Add remaining terms of p1, if any
    While (I < size1) {
        Result[k++] = p1[i++];
    }
```

```
    // Add remaining terms of p2, if any
    While (j < size2) {
        Result[k++] = p2[j++];
    }
}
// Function to display a polynomial
Void displayPolynomial(struct Term polynomial[], int size) {
    For (int I = 0; I < size; i++) {
        Printf("%dX^%dY^%d ", polynomial[i].coeff, polynomial[i].expX, polynomial[i].expY);
        If (I != size – 1) {
            Printf("+ ");
        }
    }
    Printf("\n");
}
Int main() {
    // Define the two polynomials
    Struct Term polynomial1[] = {{2, 2, 3}, {3, 1, 4}, {5, 0, 2}};
    Int size1 = sizeof(polynomial1) / sizeof(polynomial1[0]);
    Struct Term polynomial2[] = {{-2, 2, 3}, {1, 1, 4}, {4, 0, 1}};
    Int size2 = sizeof(polynomial2) / sizeof(polynomial2[0]);
    // Calculate the maximum size of the resulting polynomial
    Int maxSize = size1 + size2;
    // Create an array to store the result polynomial
    Struct Term result[maxSize];
    // Add the polynomials
    addPolynomials(polynomial1, size1, polynomial2, size2, result);
    // Display the input polynomials
    Printf("Polynomial 1: ");
    displayPolynomial(polynomial1, size1);
    printf("Polynomial 2: ");
    displayPolynomial(polynomial2, size2);
    // Display the result polynomial
    Printf("Result: ");
    displayPolynomial(result, size1 + size2);
```

```
    return 0;
}
```

This C program performs the addition of two multivariable polynomials represented as arrays of terms.

The `struct Term` represents a term in a polynomial, where `coeff` represents the coefficient, `expX` represents the exponent of the X variable, and `expY` represents the exponent of the Y variable.

The `addPolynomials` function takes two input polynomials (`p1` and `p2`) along with their sizes (`size1` and `size2`), and calculates the addition of the polynomials. It iterates over both polynomials and compares the total exponents of the terms. If a term from `p1` has a higher total exponent, it is added to the result array. If a term from `p2` has a higher total exponent, it is added to the result array. If the terms have the same total exponent, their coefficients are added, and the resulting term is added to the result array.

The `displayPolynomial` function is used to display a polynomial on the console. It iterates over the terms of the polynomial array and prints the coefficient, exponent of X, and exponent of Y. It also adds a "+" sign between terms for better readability.

In the `main` function, two input polynomials (`polynomial1` and `polynomial2`) are defined using `struct Term`. The sizes of the polynomials are calculated using the `sizeof` operator. The maximum size of the resulting polynomial is determined by adding the sizes of both input polynomials. An array called `result` is created to store the result of the addition.

The `addPolynomials` function is called to add `polynomial1` and `polynomial2`, passing the sizes and the result array. The `displayPolynomial` function is then used to display the input polynomials and the resulting polynomial on the console.

Overall, this program demonstrates how to add multivariable polynomials using explicit exponent representation and displays the result.

Write a small c program for Multiplication of two Multivariable Polynomials with two variables, using representation where the exponents are explicitly specified

```
#include <stdio.h>

#define MAX_TERMS 100

typedef struct {
    int coefficient;
    int exponent1;
    int exponent2;
} Term;

void multiplyPolynomials(Term poly1[], int size1, Term poly2[], int size2, Term result[]) {
    int i, j, k = 0;

    for (i = 0; i < size1; i++) {
        for (j = 0; j < size2; j++) {
            result[k].coefficient = poly1[i].coefficient * poly2[j].coefficient;
            result[k].exponent1 = poly1[i].exponent1 + poly2[j].exponent1;
            result[k].exponent2 = poly1[i].exponent2 + poly2[j].exponent2;
            k++;
        }
```

```
    }
}
void displayPolynomial(Term poly[], int size) {
    int i;

    for (i = 0; i < size; i++) {
        printf("(%d)x^%d*y^%d ", poly[i].coefficient, poly[i].exponent1, poly[i].exponent2);
        if (i < size - 1)
            printf("+ ");
    }

    printf("\n");
}
int main() {
    Term poly1[MAX_TERMS], poly2[MAX_TERMS], result[MAX_TERMS];
    int size1, size2, resultSize;
    printf("Enter the number of terms in polynomial 1: ");
    scanf("%d", &size1);
    printf("Enter the coefficients, exponent1, and exponent2 for each term in polynomial 1:\n");
    for (int i = 0; i < size1; i++) {
        printf("Term %d: ", i + 1);
        scanf("%d %d %d", &poly1[i].coefficient, &poly1[i].exponent1, &poly1[i].exponent2);
    }
    printf("\nEnter the number of terms in polynomial 2: ");
    scanf("%d", &size2);
    printf("Enter the coefficients, exponent1, and exponent2 for each term in polynomial 2:\n");
    for (int i = 0; i < size2; i++) {
        printf("Term %d: ", i + 1);
        scanf("%d %d %d", &poly2[i].coefficient, &poly2[i].exponent1, &poly2[i].exponent2);
    }
    multiplyPolynomials(poly1, size1, poly2, size2, result);
    resultSize = size1 * size2;
    printf("\nResultant polynomial after multiplication: ");
    displayPolynomial(result, resultSize);
    return 0;
```

}

Explain

1. We define a constant `MAX_TERMS` to represent the maximum number of terms in a polynomial.
2. We define a structure `Term` that represents a term in a polynomial. It consists of three fields: `coefficient` (the coefficient of the term), `exponent1` (the exponent of the first variable), and `exponent2` (the exponent of the second variable).
3. The `multiplyPolynomials` function takes two input polynomials (`poly1` and `poly2`), their sizes (`size1` and `size2`), and the output array for the resultant polynomial (`result`). This function multiplies each term of `poly1` with each term of `poly2` and stores the result in `result`. It does this by iterating over each term in `poly1` and `poly2` using nested loops. For each combination of terms, it multiplies the coefficients and adds the exponents to calculate the resulting term. It then stores this term in the `result` array.
4. The `displayPolynomial` function takes a polynomial `poly` and its size as input. It iterates over each term in the polynomial and prints the coefficient, exponent1, and exponent2 in the specified format. It also adds a `+` symbol between terms for better readability.
5. In the `main` function, we declare three arrays of type `Term`: `poly1`, `poly2`, and `result`, each with a size of `MAX_TERMS`. We also declare variables `size1`, `size2`, and `resultSize` to store the sizes of the polynomials.
6. The user is prompted to enter the number of terms in polynomial 1 (`size1`).
7. The program then asks the user to enter the coefficient, exponent1, and exponent2 for each term in polynomial 1 using a loop. The values are stored in the `poly1` array.
8. Similarly, the user is prompted to enter the number of terms in polynomial 2 (`size2`).
9. The program asks the user to enter the coefficient, exponent1, and exponent2 for each term in polynomial 2 using a loop. The values are stored in the `poly2` array.
10. The `multiplyPolynomials` function is called with the input polynomials `poly1` and `poly2`, their sizes `size1` and `size2`, and the `result` array to perform polynomial multiplication. The resulting polynomial will be stored in the `result` array.
11. The size of the resultant polynomial (`resultSize`) is calculated by multiplying the sizes of the input polynomials.
12. Finally, the program prints the resultant polynomial by calling the `displayPolynomial` function with the `result` array and `resultSize`.

That's how the program works! It takes two polynomials as input, multiplies them using the provided polynomial multiplication algorithm, and displays the resultant polynomial

---

Write a c program for Evaluation of Multivariable Polynomial with two variables, using representation where the exponents are explicitly specified

```
#include <stdio.h>

#include <math.h>


// Function to evaluate the polynomial

double evaluatePolynomial(double x, double y) {

  double result = 0.0;


  // Polynomial coefficients

  double coeffs[3][3] = {

    {1.0, 2.0, 3.0},

    {4.0, 5.0, 6.0},

    {7.0, 8.0, 9.0}

  };

  // Exponents corresponding to the variables x and y

  int exponents[3][3] = {

    {2, 1, 0},
```

```
    {1, 0, 1},
    {0, 1, 2}
  };
  // Evaluate the polynomial
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      result += coeffs[i][j] * pow(x, exponents[i][j]) * pow(y, exponents[i][j]);
    }
  }
  return result;
}
int main() {
  double x, y;
  printf("Enter the value of x: ");
  scanf("%lf", &x);
  printf("Enter the value of y: ");
  scanf("%lf", &y);
  double result = evaluatePolynomial(x, y);
  printf("Result: %lf\n", result);
  return 0;
}
```

Explain

1. The program includes the necessary header files: `<stdio.h>` for input/output operations and `<math.h>` for mathematical functions.
2. The `evaluatePolynomial` function takes two variables `x` and `y` as input and returns the evaluated result of the polynomial. It initializes the `result` variable to 0.0.
3. The polynomial coefficients and exponents are defined using 2D arrays. In this example, the polynomial has 3 terms, and each term has coefficients and exponents for `x` and `y`. The coefficients are stored in the `coeffs` array, and the exponents are stored in the `exponents` array.
4. The polynomial is evaluated using a nested loop. The outer loop iterates over the rows of the `coeffs` and `exponents` arrays, representing each term of the polynomial. The inner loop iterates over the columns, representing the coefficients and exponents for `x` and `y` within each term. The `result` is updated by multiplying each coefficient with the respective powers of `x` and `y` using the `pow` function from the `math.h` library.
5. The `main` function is the entry point of the program. It prompts the user to enter the values of `x` and `y` using `printf` and reads the input using `scanf`.
6. The `evaluatePolynomial` function is called with the values of `x` and `y` to compute the result.
7. Finally, the result is printed on the console using `printf`.

This code allows you to easily modify the coefficients and exponents arrays to represent different multivariable polynomials.

Write a c program for database of students (name, phone number, address, 10^{th} marks, 12^{th} marks, eligible for engineering or not)

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

```
struct Student {
    char name[50];
    char phoneNumber[15];
    char address[100];
    float tenthMarks;
    float twelfthMarks;
    char eligibility[4];
};
void clearInputBuffer() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF);
}

void printStudent(struct Student student) {
    printf("Name: %s\n", student.name);
    printf("Phone Number: %s\n", student.phoneNumber);
    printf("Address: %s\n", student.address);
    printf("10th Marks: %.2f\n", student.tenthMarks);
    printf("12th Marks: %.2f\n", student.twelfthMarks);
    printf("Eligible for Engineering: %s\n", student.eligibility);
    printf("\n");
}

int main() {
    int numStudents, i;

    printf("Enter the number of students: ");
    scanf("%d", &numStudents);
    clearInputBuffer();

    struct Student* students = malloc(numStudents * sizeof(struct Student));

    for (i = 0; i < numStudents; i++) {
        printf("\nStudent %d\n", i + 1);
```

```
        printf("Enter name: ");
        fgets(students[i].name, sizeof(students[i].name), stdin);
        students[i].name[strcspn(students[i].name, "\n")] = '\0';

        printf("Enter phone number: ");
        fgets(students[i].phoneNumber, sizeof(students[i].phoneNumber), stdin);
        students[i].phoneNumber[strcspn(students[i].phoneNumber, "\n")] = '\0';

        printf("Enter address: ");
        fgets(students[i].address, sizeof(students[i].address), stdin);
        students[i].address[strcspn(students[i].address, "\n")] = '\0';

        printf("Enter 10th marks: ");
        scanf("%f", &students[i].tenthMarks);
        clearInputBuffer();

        printf("Enter 12th marks: ");
        scanf("%f", &students[i].twelfthMarks);
        clearInputBuffer();
        if (students[i].tenthMarks >= 60 && students[i].twelfthMarks >= 60) {
            strcpy(students[i].eligibility, "Yes");
        } else {
            strcpy(students[i].eligibility, "No");
        }
    }
    printf("\n--- Student Database ---\n\n");
    for (i = 0; i < numStudents; i++) {
        printf("Student %d\n", i + 1);
        printStudent(students[i]);
    }
    free(students);
    return 0;
}
```

Explain

1. We begin by including necessary header files: `<stdio.h>` for input/output operations and `<stdlib.h>` for memory allocation with `malloc` and `free`.
2. Next, we define a structure named `Student` that represents the information of a student. It contains fields for the student's name, phone number, address, 10th marks, 12th marks, and eligibility.
3. The `clearInputBuffer()` function is a utility function used to clear any remaining characters in the input buffer after a user enters data.
4. The `printStudent()` function is responsible for displaying the details of a student.
5. In the `main()` function, we prompt the user to enter the number of students and store it in the `numStudents` variable.
6. We allocate memory dynamically using `malloc` to create an array of `Student` structures based on the number of students entered.
7. A `for` loop is used to iterate over each student. Inside the loop, the program prompts the user to enter details for each student, such as name, phone number, address, 10th marks, and 12th marks.
8. The `fgets` function is used to read input for string fields (name, phone number, and address) to handle spaces and avoid buffer overflow. The `strcspn` function is used to remove the newline character from the input string.
9. The `scanf` function is used to read floating-point numbers for the 10th and 12th marks.
10. After inputting the marks, the program checks if the student is eligible for engineering based on the minimum marks requirement (60% or above for both 10th and 12th). If the student meets the eligibility criteria, the eligibility field is set to "Yes"; otherwise, it is set to "No".
11. Once all the student details are entered and eligibility is determined, the program proceeds to display the student database.
12. The loop iterates over each student and calls the `printStudent()` function to display the details.
13. Finally, the program frees the dynamically allocated memory using the `free()` function.

That's a brief explanation of the program's functionality. It allows the user to input student information, calculates their eligibility for engineering, and displays the student database with all the details.

Write a c program for database of player (any game)

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_PLAYERS 100


Typedef struct {

   Char name[50];

   Int score;

} Player;


Typedef struct {

   Player players[MAX_PLAYERS];

   Int count;

} PlayerDatabase;


Void initializeDatabase(PlayerDatabase *database) {

   Database->count = 0;

}
```

```
Void addPlayer(PlayerDatabase *database, const char *name, int score) {
   If (database->count < MAX_PLAYERS) {
      Player *player = &(database->players[database->count]);
      Strncpy(player->name, name, sizeof(player->name) – 1);
      Player->score = score;
      Database->count++;
      Printf(“Player added successfully.\n”);
   } else {
      Printf(“Database is full. Cannot add more players.\n”);
   }
}

Void displayPlayers(PlayerDatabase *database) {
   Printf(“Player Database:\n”);
   Printf(“----------\n”);
   For (int I = 0; I < database->count; i++) {
      Player *player = &(database->players[i]);
      Printf(“Name: %s\n”, player->name);
      Printf(“Score: %d\n”, player->score);
      Printf(“----------\n”);
   }
}

Int main() {
   PlayerDatabase database;
   initializeDatabase(&database);

   int choice;
   char name[50];
   int score;

   do {
      printf(“Menu\n”);
      printf(“1. Add player\n”);
```

```
    printf("2. Display players\n");
    printf("3. Quit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
      case 1:
        printf("Enter player name: ");
        scanf("%s", name);
        printf("Enter player score: ");
        scanf("%d", &score);
        addPlayer(&database, name, score);
        break;
      case 2:
        displayPlayers(&database);
        break;
      case 3:
        printf("Exiting program.\n");
        break;
      default:
        printf("Invalid choice. Please try again.\n");
        break;
    }
  } while (choice != 3);
  Return 0;
}
```

1. We include the necessary header files: `stdio.h` for input/output operations, `stdlib.h` for memory allocation, and `string.h` for string manipulation functions.
2. We define a constant `MAX_PLAYERS` to specify the maximum number of players that can be stored in the database.
3. We define a `Player` structure to hold the information of a player. It consists of a character array `name` to store the player's name and an integer `score` to store the player's score.
4. We define a `PlayerDatabase` structure to hold the array of players and a counter `count` to keep track of the number of players in the database.
5. The `initializeDatabase` function initializes the player database by setting the `count` to zero.
6. The `addPlayer` function takes the player's name and score as input and adds a new player to the database if there is space available. It first checks if the database is not already full. If there is space, it retrieves the next available index, copies the name using `strncpy` (ensuring it fits within the allocated space), assigns the score, increments the `count`, and prints a success message. If the database is full, it prints an error message.
7. The `displayPlayers` function iterates over the players in the database and prints their names and scores.
8. In the `main` function, we declare a `PlayerDatabase` object called `database` and initialize it using `initializeDatabase`.
9. The program enters a do-while loop that displays a menu and reads the user's choice using `scanf`.

10. Depending on the choice, the program either prompts the user for the player's name and score and calls the `addPlayer` function, or it calls the `displayPlayers` function to show all the players in the database. If the choice is to quit, the program prints a message and exits the loop.
11. The loop continues until the user chooses to quit (choice 3).
12. Finally, the program returns 0 to indicate successful execution.

The program allows the user to add players to the database by entering their names and scores, display the list of players, and quit the program when desired. The player data is stored in memory during runtime and is lost once the program terminates.

You can customize and extend this program further based on your specific requirements.

Write a c program for database of colleges/universities (name, address, status wrt NAAC, NIRF, NBA

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_COLLEGES 100


typedef struct {

   char name[100];

   char address[100];

   char naacStatus[20];

   char nirfStatus[20];

   char nbaStatus[20];

} College;


typedef struct {

   College colleges[MAX_COLLEGES];

   int count;

} CollegeDatabase;


void initializeDatabase(CollegeDatabase *database) {

   database->count = 0;

}


void addCollege(CollegeDatabase *database, const char *name, const char *address, const char *naacStatus, const char *nirfStatus, const char *nbaStatus) {

   if (database->count < MAX_COLLEGES) {

      College *college = &(database->colleges[database->count]);

      strncpy(college->name, name, sizeof(college->name) - 1);
```

```
        strncpy(college->address, address, sizeof(college->address) - 1);
        strncpy(college->naacStatus, naacStatus, sizeof(college->naacStatus) - 1);
        strncpy(college->nirfStatus, nirfStatus, sizeof(college->nirfStatus) - 1);
        strncpy(college->nbaStatus, nbaStatus, sizeof(college->nbaStatus) - 1);
        database->count++;
        printf("College added successfully.\n");
    } else {
        printf("Database is full. Cannot add more colleges.\n");
    }
}
void displayColleges(CollegeDatabase *database) {
    printf("College Database:\n");
    printf("-----------------\n");
    for (int i = 0; i < database->count; i++) {
        College *college = &(database->colleges[i]);
        printf("Name: %s\n", college->name);
        printf("Address: %s\n", college->address);
        printf("NAAC Status: %s\n", college->naacStatus);
        printf("NIRF Status: %s\n", college->nirfStatus);
        printf("NBA Status: %s\n", college->nbaStatus);
        printf("-----------------\n");
    }
}
int main() {
    CollegeDatabase database;
    initializeDatabase(&database);

    int choice;
    char name[100];
    char address[100];
    char naacStatus[20];
    char nirfStatus[20];
    char nbaStatus[20];

    do {
```

```
        printf("Menu\n");
        printf("1. Add college\n");
        printf("2. Display colleges\n");
        printf("3. Quit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter college name: ");
                scanf("%s", name);
                printf("Enter college address: ");
                scanf("%s", address);
                printf("Enter NAAC status: ");
                scanf("%s", naacStatus);
                printf("Enter NIRF status: ");
                scanf("%s", nirfStatus);
                printf("Enter NBA status: ");
                scanf("%s", nbaStatus);
                addCollege(&database, name, address, naacStatus, nirfStatus, nbaStatus);
                break;
            case 2:
                displayColleges(&database);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
                break;
        }
    } while (choice != 3);
    return 0;
}
```

Explain

1. We define a `College` structure to hold the information of a college or university. It consists of character arrays for name, address, NAAC status, NIRF status, and NBA status.
2. We define a `CollegeDatabase` structure to hold the array of colleges and a counter `count` to keep track of the number of colleges in the database.
3. The `initializeDatabase` function initializes the college database by setting the `count` to zero.
4. The `addCollege` function takes the college's name, address, NAAC status, NIRF status, and NBA status as input and adds a new college to the database if there is space available. It first checks if the database is not already full. If there is space, it retrieves the next available index, copies the provided information using `strncpy` (ensuring they fit within the allocated space), increments the `count`, and prints a success message. If the database is full, it prints an error message.
5. The `displayColleges` function iterates over the colleges in the database and prints their name, address, NAAC status, NIRF status, and NBA status.
6. In the `main` function, a `CollegeDatabase` object called `database` is created and initialized using `initializeDatabase`.
7. The program enters a do-while loop that displays a menu and reads the user's choice using `scanf`.
8. Depending on the choice, the program either prompts the user for the college's name, address, NAAC status, NIRF status, and NBA status and calls the `addCollege` function, or it calls the `displayColleges` function to show all the colleges in the database. If the choice is to quit, the program prints a message and exits the loop.
9. The loop continues until the user chooses to quit (choice 3).
10. Finally, the program returns 0 to indicate successful execution.

Write a c program with suitable logic for database of weather (precipitation, humidity, temp, region)

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_RECORDS 100


typedef struct {

   double precipitation;

   int humidity;

   double temperature;

   char region[50];

} WeatherRecord;


typedef struct {

   WeatherRecord records[MAX_RECORDS];

   int count;

} WeatherDatabase;


void initializeDatabase(WeatherDatabase *database) {

   database->count = 0;

}


void addRecord(WeatherDatabase *database, double precipitation, int humidity, double temperature, const char *region) {
```

```
    if (database->count < MAX_RECORDS) {
        WeatherRecord *record = &(database->records[database->count]);
        record->precipitation = precipitation;
        record->humidity = humidity;
        record->temperature = temperature;
        strncpy(record->region, region, sizeof(record->region) - 1);
        database->count++;
        printf("Weather record added successfully.\n");
    } else {
        printf("Database is full. Cannot add more records.\n");
    }
}

void displayRecords(WeatherDatabase *database) {
    printf("Weather Database:\n");
    printf("-----------------\n");
    for (int i = 0; i < database->count; i++) {
        WeatherRecord *record = &(database->records[i]);
        printf("Precipitation: %.2lf\n", record->precipitation);
        printf("Humidity: %d\n", record->humidity);
        printf("Temperature: %.2lf\n", record->temperature);
        printf("Region: %s\n", record->region);
        printf("-----------------\n");
    }
}
int main() {
    WeatherDatabase database;
    initializeDatabase(&database);

    int choice;
    double precipitation;
    int humidity;
    double temperature;
    char region[50];
```

```
    do {
        printf("Menu\n");
        printf("1. Add weather record\n");
        printf("2. Display weather records\n");
        printf("3. Quit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter precipitation: ");
                scanf("%lf", &precipitation);
                printf("Enter humidity: ");
                scanf("%d", &humidity);
                printf("Enter temperature: ");
                scanf("%lf", &temperature);
                printf("Enter region: ");
                scanf("%s", region);
                addRecord(&database, precipitation, humidity, temperature, region);
                break;
            case 2:
                displayRecords(&database);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
                break;
        }
    } while (choice != 3);
    return 0;
}
```

The updated code is a C program that manages a database of weather records. It allows the user to add weather records with precipitation, humidity, temperature, and region, as well as display all the weather records stored in the database.

program's logic:

1. The program defines a `WeatherRecord` structure to hold weather information for a specific region. It includes variables for precipitation (double), humidity (integer), temperature (double), and region (character array).
2. A `WeatherDatabase` structure is defined to hold an array of `WeatherRecord` structures and a count variable to keep track of the number of records in the database.
3. The `initializeDatabase` function initializes the weather database by setting the count to zero.
4. The `addRecord` function takes the precipitation, humidity, temperature, and region as input parameters. It adds a new weather record to the database if there is space available. If there is enough space, it retrieves the next available index, assigns the provided values to the corresponding fields in the `WeatherRecord` structure, increments the count, and displays a success message. If the database is full, it displays an error message.
5. The `displayRecords` function prints all the weather records stored in the database. It iterates over each record and prints its precipitation, humidity, temperature, and region.
6. In the `main` function, a `WeatherDatabase` object called `database` is created and initialized using the `initializeDatabase` function.
7. The program enters a do-while loop, displaying a menu of options and reading the user's choice using `scanf`.
8. Depending on the user's choice, the program either prompts the user for the precipitation, humidity, temperature, and region to add a new weather record using the `addRecord` function, or it displays all the weather records in the database using the `displayRecords` function. If the user chooses to quit, the program prints a message and exits the loop.
9. The loop continues until the user chooses to quit (choice 3).
10. Finally, the program returns 0 to indicate successful execution.

6.Database of non- teaching staff (name, age, id, contact, address, profile)

7. database of startups (company name, founder, category/field, net worth)

8. database of bollywood (name, age, gender, profile, networth)

9. database of politicians (name, age, gender, profile, networth, region, social status)

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_ENTRIES 100


// Structs for different databases

Typedef struct {

   Char name[50];

   Int age;

   Int id;

   Char contact[20];

   Char address[100];

   Char profile[100];

} NonTeachingStaff;


Typedef struct {

   Char companyName[50];

   Char founder[50];
```

```
    Char category[50];
    Double netWorth;
} Startup;

Typedef struct {
    Char name[50];
    Int age;
    Char gender;
    Char profile[100];
    Double netWorth;
} Bollywood;

Typedef struct {
    Char name[50];
    Int age;
    Char gender;
    Char profile[100];
    Double netWorth;
    Char region[50];
    Char socialStatus[50];
} Politician;

// Function prototypes
Void addNonTeachingStaff(NonTeachingStaff staff[], int *count);
Void addStartup(Startup startups[], int *count);
Void addBollywood(Bollywood bollywood[], int *count);
Void addPolitician(Politician politicians[], int *count);
Void displayNonTeachingStaff(NonTeachingStaff staff[], int count);
Void displayStartups(Startup startups[], int count);
Void displayBollywood(Bollywood bollywood[], int count);
Void displayPoliticians(Politician politicians[], int count);

Int main() {
    Int choice;
    NonTeachingStaff nonTeachingStaff[MAX_ENTRIES];
```

```
Startup startups[MAX_ENTRIES];
Bollywood bollywood[MAX_ENTRIES];
Politician politicians[MAX_ENTRIES];
Int nonTeachingCount = 0;
Int startupCount = 0;
Int bollywoodCount = 0;
Int politicianCount = 0;

Do {
   Printf("1. Add Non-Teaching Staff\n");
   Printf("2. Add Startup\n");
   Printf("3. Add Bollywood Entry\n");
   Printf("4. Add Politician\n");
   Printf("5. Display Non-Teaching Staff\n");
   Printf("6. Display Startups\n");
   Printf("7. Display Bollywood Entries\n");
   Printf("8. Display Politicians\n");
   Printf("9. Exit\n");
   Printf("Enter your choice: ");
   Scanf("%d", &choice);

   Switch (choice) {
      Case 1:
         addNonTeachingStaff(nonTeachingStaff, &nonTeachingCount);
         break;
      case 2:
         addStartup(startups, &startupCount);
         break;
      case 3:
         addBollywood(bollywood, &bollywoodCount);
         break;
      case 4:
         addPolitician(politicians, &politicianCount);
         break;
      case 5:
```

```
            displayNonTeachingStaff(nonTeachingStaff, nonTeachingCount);
            break;
        case 6:
            displayStartups(startups, startupCount);
            break;
        case 7:
            displayBollywood(bollywood, bollywoodCount);
            break;
        case 8:
            displayPoliticians(politicians, politicianCount);
            break;
        case 9:
            printf("Exiting program...\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
            break;
    }

    Printf("\n");
  } while (choice != 9);

  Return 0;
}

Void addNonTeachingStaff(NonTeachingStaff staff[], int *count) {
  If (*count == MAX_ENTRIES) {
    Printf("Database is full. Cannot add more entries.\n");
    Return;
  }

  Printf("Enter name: ");
  Scanf("%s", staff[*count].name);
  Printf("Enter age: ");
  Scanf("%d", &staff[*count].age);
```

```
    Printf(“Enter ID: “);
    Scanf(“%d”, &staff[*count].id);
    Printf(“Enter contact number: “);
    Scanf(“%s”, staff[*count].contact);
    Printf(“Enter address: “);
    Scanf(“%s”, staff[*count].address);
    Printf(“Enter profile: “);
    Scanf(“%s”, staff[*count].profile);

    (*count)++;
}

Void addStartup(Startup startups[], int *count) {
    If (*count == MAX_ENTRIES) {
        Printf(“Database is full. Cannot add more entries.\n”);
        Return;
    }

    Printf(“Enter company name: “);
    Scanf(“%s”, startups[*count].companyName);
    Printf(“Enter founder name: “);
    Scanf(“%s”, startups[*count].founder);
    Printf(“Enter category/field: “);
    Scanf(“%s”, startups[*count].category);
    Printf(“Enter net worth: “);
    Scanf(“%lf”, &startups[*count].netWorth);

    (*count)++;
}

Void addBollywood(Bollywood bollywood[], int *count) {
    If (*count == MAX_ENTRIES) {
        Printf(“Database is full. Cannot add more entries.\n”);
        Return;
    }
```

```
    Printf(“Enter name: “);
    Scanf(“%s”, bollywood[*count].name);
    Printf(“Enter age: “);
    Scanf(“%d”, &bollywood[*count].age);
    Printf(“Enter gender (M/F): “);
    Scanf(“ %c”, &bollywood[*count].gender);
    Printf(“Enter profile: “);
    Scanf(“%s”, bollywood[*count].profile);
    Printf(“Enter net worth: “);
    Scanf(“%lf”, &bollywood[*count].netWorth);

    (*count)++;
}

Void addPolitician(Politician politicians[], int *count) {
    If (*count == MAX_ENTRIES) {
        Printf(“Database is full. Cannot add more entries.\n”);
        Return;
    }

    Printf(“Enter name: “);
    Scanf(“%s”, politicians[*count].name);
    Printf(“Enter age: “);
    Scanf(“%d”, &politicians[*count].age);
    Printf(“Enter gender (M/F): “);
    Scanf(“ %c”, &politicians[*count].gender);
    Printf(“Enter profile: “);
    Scanf(“%s”, politicians[*count].profile);
    Printf(“Enter net worth: “);
    Scanf(“%lf”, &politicians[*count].netWorth);
    Printf(“Enter region: “);
    Scanf(“%s”, politicians[*count].region);
    Printf(“Enter social status: “);
    Scanf(“%s”, politicians[*count].socialStatus);
```

```
    (*count)++;
}

Void displayNonTeachingStaff(NonTeachingStaff staff[], int count) {
    If (count == 0) {
        Printf(“No non-teaching staff entries found.\n”);
        Return;
    }

    Printf(“Non-Teaching Staff Database:\n”);
    For (int I = 0; I < count; i++) {
        Printf(“Entry %d:\n”, I + 1);
        Printf(“Name: %s\n”, staff[i].name);
        Printf(“Age: %d\n”, staff[i].age);
        Printf(“ID: %d\n”, staff[i].id);
        Printf(“Contact: %s\n”, staff[i].contact);
        Printf(“Address: %s\n”, staff[i].address);
        Printf(“Profile: %s\n”, staff[i].profile);
        Printf(“\n”);
    }
}

Void displayStartups(Startup startups[], int count) {
    If (count == 0) {
        Printf(“No startup entries found.\n”);
        Return;
    }

    Printf(“Startup Database:\n”);
    For (int I = 0; I < count; i++) {
        Printf(“Entry %d:\n”, I + 1);
        Printf(“Company Name: %s\n”, startups[i].companyName);
        Printf(“Founder: %s\n”, startups[i].founder);
        Printf(“Category/Field: %s\n”, startups[i].category);
```

```
        Printf(“Net Worth: %.2lf\n”, startups[i].netWorth);
        Printf(“\n”);
    }
}

Void displayBollywood(Bollywood bollywood[], int count) {
    If (count == 0) {
        Printf(“No Bollywood entries found.\n”);
        Return;
    }

    Printf(“Bollywood Database:\n”);
    For (int I = 0; I < count; i++) {
        Printf(“Entry %d:\n”, I + 1);
        Printf(“Name: %s\n”, bollywood[i].name);
        Printf(“Age: %d\n”, bollywood[i].age);
        Printf(“Gender: %c\n”, bollywood[i].gender);
        Printf(“Profile: %s\n”, bollywood[i].profile);
        Printf(“Net Worth: %.2lf\n”, bollywood[i].netWorth);
        Printf(“\n”);
    }
}

Void displayPoliticians(Politician politicians[], int count) {
    If (count == 0) {
        Printf(“No politician entries found.\n”);
        Return;
    }

    Printf(“Politician Database:\n”);
    For (int I = 0; I < count; i++) {
        Printf(“Entry %d:\n”, I + 1);
        Printf(“Name: %s\n”, politicians[i].name);
        Printf(“Age: %d\n”, politicians[i].age);
        Printf(“Gender: %c\n”, politicians[i].gender);
```

```
        Printf("Profile: %s\n", politicians[i].profile);
        Printf("Net Worth: %.2lf\n", politicians[i].netWorth);
        Printf("Region: %s\n", politicians[i].region);
        Printf("Social Status: %s\n", politicians[i].socialStatus);
        Printf("\n");
    }
}
```

Write a c program for database of electricity utilization of each building (building no, room no and total number of rooms, number of lights and fans, number of AC's (company, star rating, capacity), projector) – one building one group

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

Struct AC {
    Char company[20];
    Int star_rating;
    Int capacity;
};

Struct Building {
    Int building_no;
    Int room_no;
    Int total_rooms;
    Int num_lights;
    Int num_fans;
    Struct AC ac;
    Int has_projector;
};

Void displayBuilding(struct Building b) {
    Printf("Building Number: %d\n", b.building_no);
    Printf("Room Number: %d\n", b.room_no);
    Printf("Total Rooms: %d\n", b.total_rooms);
    Printf("Number of Lights: %d\n", b.num_lights);
    Printf("Number of Fans: %d\n", b.num_fans);
    Printf("AC Details:\n");
```

```
    Printf("   - Company: %s\n", b.ac.company);
    Printf("   - Star Rating: %d\n", b.ac.star_rating);
    Printf("   - Capacity: %d\n", b.ac.capacity);
    Printf("Has Projector: %s\n", b.has_projector ? "Yes" : "No");
    Printf("\n");
}

Int main() {
    Int num_buildings;

    Printf("Enter the number of buildings: ");
    Scanf("%d", &num_buildings);

    Struct Building* buildings = (struct Building*)malloc(num_buildings * sizeof(struct Building));

    For (int I = 0; I < num_buildings; i++) {
        Printf("\nBuilding %d Details:\n", I + 1);

        Printf("Enter Building Number: ");
        Scanf("%d", &buildings[i].building_no);

        Printf("Enter Room Number: ");
        Scanf("%d", &buildings[i].room_no);

        Printf("Enter Total Rooms: ");
        Scanf("%d", &buildings[i].total_rooms);

        Printf("Enter Number of Lights: ");
        Scanf("%d", &buildings[i].num_lights);

        Printf("Enter Number of Fans: ");
        Scanf("%d", &buildings[i].num_fans);

        Printf("Enter AC Details:\n");
        Printf("   - Enter Company: ");
```

```
    Scanf("%s", buildings[i].ac.company);

    Printf("   - Enter Star Rating: ");
    Scanf("%d", &buildings[i].ac.star_rating);

    Printf("   - Enter Capacity: ");
    Scanf("%d", &buildings[i].ac.capacity);

    Printf("Does it have a Projector? (1 for Yes / 0 for No): ");
    Scanf("%d", &buildings[i].has_projector);
  }

  Printf("\n--- Building Database ---\n\n");

  For (int I = 0; I < num_buildings; i++) {
    Printf("Building %d:\n", I + 1);
    displayBuilding(buildings[i]);
  }

  Free(buildings);

  Return 0;
}
```

Explain

1. The code begins by including the necessary header files, `stdio.h` and `stdlib.h`, which provide functions for input/output and memory allocation, respectively.
2. Two structures are defined: `AC` and `Building`. The `AC` structure represents the details of an air conditioner, including the company name, star rating, and capacity. The `Building` structure represents the details of a building, including the building number, room number, total rooms, number of lights and fans, AC details (represented by the `AC` structure), and the presence of a projector.
3. The `displayBuilding` function takes a `Building` structure as an argument and prints out its details in a formatted manner.
4. In the `main` function, the user is prompted to enter the number of buildings in the database.
5. Memory is dynamically allocated to store an array of `Building` structures based on the number of buildings entered by the user.
6. A `for` loop is used to iterate through each building and collect its details from the user using `scanf` statements. The user is prompted to enter the building number, room number, total rooms, number of lights, number of fans, AC details (company, star rating, and capacity), and whether the building has a projector.
7. After collecting the building details, the `displayBuilding` function is called for each building to display its information.
8. Finally, the memory allocated for the array of `Building` structures is freed using the `free` function.

The code allows the user to create a database of electricity utilization for multiple buildings, with each building having its own set of attributes and details.

Write a c program for database of IPL (team name, networth, owner of the team, captain and VC, matches won last

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

#define MAX_TEAMS 10
#define MAX_NAME_LENGTH 50

Struct IPLTeam {
   Char name[MAX_NAME_LENGTH];
   Float networth;
   Char owner[MAX_NAME_LENGTH];
   Char captain[MAX_NAME_LENGTH];
   Char viceCaptain[MAX_NAME_LENGTH];
   Int matchesWon;
};

Void displayTeam(struct IPLTeam team) {
   Printf("Team Name: %s\n", team.name);
   Printf("Net Worth: %.2f million\n", team.networth);
   Printf("Owner: %s\n", team.owner);
   Printf("Captain: %s\n", team.captain);
   Printf("Vice Captain: %s\n", team.viceCaptain);
   Printf("Matches Won Last: %d\n\n", team.matchesWon);
}

Int main() {
   Struct IPLTeam teams[MAX_TEAMS];
   Int numTeams = 0;

   While (numTeams < MAX_TEAMS) {
      Struct IPLTeam newTeam;

      Printf("Enter details for Team %d:\n", numTeams + 1);

      Printf("Team Name: ");
```

```
    Fgets(newTeam.name, MAX_NAME_LENGTH, stdin);

    Printf("Net Worth (in million): ");
    Scanf("%f", &newTeam.networth);
    Getchar();  // Clear the newline character from input buffer

    Printf("Owner: ");
    Fgets(newTeam.owner, MAX_NAME_LENGTH, stdin);

    Printf("Captain: ");
    Fgets(newTeam.captain, MAX_NAME_LENGTH, stdin);

    Printf("Vice Captain: ");
    Fgets(newTeam.viceCaptain, MAX_NAME_LENGTH, stdin);

    Printf("Matches Won Last: ");
    Scanf("%d", &newTeam.matchesWon);
    Getchar();  // Clear the newline character from input buffer

    Teams[numTeams] = newTeam;
    numTeams++;

    printf("\n");

    char choice;
    printf("Do you want to enter details for another team? (y/n): ");
    scanf("%c", &choice);
    getchar();  // Clear the newline character from input buffer
    if (choice != 'y' && choice != 'Y') {
        break;
    }
}
Printf("\n--- IPL Team Database ---\n\n");
For (int I = 0; I < numTeams; i++) {
    Printf("Details for Team %d:\n", I + 1);
```

```
    displayTeam(teams[i]);
  }
  Return 0;
}
```

This program is a simple database for IPL teams. Let's go through the code and explain its functionality:

1. The program begins by including the necessary header files: `<stdio.h>`, `<stdlib.h>`, and `<string.h>`. These headers provide functions for input/output operations, dynamic memory allocation, and string manipulation.

2. Next, the program defines some constants:

- `MAX_TEAMS` represents the maximum number of teams that can be stored in the database.

- `MAX_NAME_LENGTH` represents the maximum length of the team name, owner name, captain name, and vice-captain name.

2. The program declares a structure `struct IPLTeam` to represent an IPL team. It contains fields for the team name, net worth, owner, captain, vice-captain, and matches won last.
3. The `displayTeam` function is defined to display the details of a team. It takes a `struct IPLTeam` as a parameter and prints the team's information.
4. In the `main` function, an array of `struct IPLTeam` called `teams` is declared to store the teams' data. The variable `numTeams` keeps track of the number of teams entered.
5. The program uses a `while` loop to repeatedly prompt the user to enter details for each team until the maximum number of teams is reached or the user decides not to enter more details.
6. Inside the loop, a new team is created using a temporary `struct IPLTeam` variable called `newTeam`.
7. The user is prompted to enter the details for the team, such as the team name, net worth, owner, captain, vice-captain, and matches won last. Input functions like `fgets` and `scanf` are used to read the user's input. The `getchar` function is called to clear the newline character from the input buffer.
8. The newly created `newTeam` is then added to the `teams` array, and the `numTeams` counter is incremented.
9. After entering the details for a team, the user is asked whether they want to enter details for another team. If the user enters 'y' or 'Y', the loop continues; otherwise, it breaks.
10. Once the user finishes entering the team details, the program displays the IPL Team Database by iterating over the `teams` array and calling the `displayTeam` function for each team.
11. Finally, the program returns 0 to indicate successful execution.

This program allows the user to create a database of IPL teams by entering their details and then displays the information for each team.

Write a c program for Database of work of ISRO – (name of activity, scientist involved, advantage to society, minimum 10 such activities

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_ACTIVITIES 10

#define MAX_NAME_LENGTH 50

#define MAX_SCIENTISTS 5

#define MAX_ADVANTAGE_LENGTH 100


struct Activity {

  char name[MAX_NAME_LENGTH];

  char scientists[MAX_SCIENTISTS][MAX_NAME_LENGTH];

  char advantage[MAX_ADVANTAGE_LENGTH];
```

```
};

void addActivity(struct Activity activities[], int *count) {
    if (*count >= MAX_ACTIVITIES) {
        printf("The database is full. Cannot add more activities.\n");
        return;
    }

    struct Activity newActivity;
    printf("Enter the name of the activity: ");
    fgets(newActivity.name, MAX_NAME_LENGTH, stdin);
    newActivity.name[strcspn(newActivity.name, "\n")] = '\0';

    printf("Enter the names of scientists involved (comma-separated): ");
    fgets(newActivity.scientists[0], MAX_NAME_LENGTH, stdin);
    newActivity.scientists[0][strcspn(newActivity.scientists[0], "\n")] = '\0';

    char* scientistName = strtok(newActivity.scientists[0], ",");
    int i = 1;
    while (scientistName != NULL && i < MAX_SCIENTISTS) {
        strcpy(newActivity.scientists[i], scientistName);
        scientistName = strtok(NULL, ",");
        i++;
    }

    printf("Enter the advantage to society: ");
    fgets(newActivity.advantage, MAX_ADVANTAGE_LENGTH, stdin);
    newActivity.advantage[strcspn(newActivity.advantage, "\n")] = '\0';

    activities[*count] = newActivity;
    (*count)++;
    printf("Activity added successfully!\n");
}

void displayActivities(struct Activity activities[], int count) {
```

```
    if (count == 0) {
        printf("No activities found in the database.\n");
        return;
    }

    printf("Activities:\n");
    for (int i = 0; i < count; i++) {
        printf("Activity %d:\n", i + 1);
        printf("Name: %s\n", activities[i].name);
        printf("Scientists involved: ");
        for (int j = 0; j < MAX_SCIENTISTS && activities[i].scientists[j][0] != '\0'; j++) {
            printf("%s, ", activities[i].scientists[j]);
        }
        printf("\n");
        printf("Advantage to society: %s\n", activities[i].advantage);
        printf("\n");
    }
}

int main() {
    struct Activity activities[MAX_ACTIVITIES];
    int count = 0;
    int choice;

    do {
        printf("********** ISRO Activity Database **********\n");
        printf("1. Add an activity\n");
        printf("2. Display all activities\n");
        printf("3. Quit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar(); // Consume newline character

        switch (choice) {
            case 1:
```

```
            addActivity(activities, &count);
            break;
        case 2:
            displayActivities(activities, count);
            break;
        case 3:
            printf("Exiting program. Goodbye!\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
    }
    printf("\n");
  } while (choice != 3);
  return 0;
}
```

Explain

1. The code begins by including the necessary header files: `<stdio.h>` for input/output operations, `<stdlib.h>` for dynamic memory allocation, and `<string.h>` for string manipulation functions.
2. The program defines several constants to control the size of the activity database and the maximum length of activity names, scientist names, and advantages.
3. A structure called `Activity` is defined to represent an activity in the database. It contains three members: `name` (to store the name of the activity), `scientists` (a 2D array to store the names of scientists involved), and `advantage` (to store the advantage of the activity to society).
4. The program defines two functions: `addActivity` and `displayActivities`.
5. The `addActivity` function allows the user to add a new activity to the database. It checks if the database is already full and returns an error message if it is. Otherwise, it prompts the user to enter the name of the activity, names of scientists involved (comma-separated), and the advantage of the activity. It stores the entered data in a new `Activity` structure and adds it to the `activities` array. The `count` parameter keeps track of the number of activities in the database.
6. The `displayActivities` function is used to display all the activities stored in the `activities` array. It checks if there are any activities in the database and displays each activity's details, including the name, scientists involved, and advantage.
7. In the `main` function, an array of `Activity` structures called `activities` is created to hold the activities. The `count` variable keeps track of the number of activities in the database. The `choice` variable is used to store the user's menu choice.
8. The program enters a do-while loop to display the menu and process the user's choice until the user chooses to quit.
9. Inside the loop, the menu options are displayed, and the user's choice is read using `scanf`. The `getchar` function is used to consume the newline character after reading the choice.
10. Based on the user's choice, the corresponding case in the switch statement is executed. Case 1 calls the `addActivity` function to add a new activity, case 2 calls the `displayActivities` function to display all activities, and case 3 displays a goodbye message and breaks out of the loop.
11. If the user enters an invalid choice, a default case is executed to display an error message.
12. After executing the selected option, the program continues to the top of the loop, displaying the menu again, and waiting for the user to enter another choice.
13. The loop continues until the user chooses to quit (choice 3), at which point the program exits.

That's the overview of the provided code. It allows you to interactively add activities to the database and display them.