Addition of sparse matrix

```
#include<stdio.h>

Int main()
{
    Int n, m, c, d, matrix[10][10];
    Int counter = 0;

    Printf("Enter the number of rows and columns of the matrix: ");
    Scanf("%d%d", &m, &n);

    Printf("Enter the %d elements of the matrix:\n", m * n);
    For(c = 0; c < m; c++)   // to iterate the rows
    {
        For(d = 0; d < n; d++)   // to iterate the columns
        {
            Scanf("%d", &matrix[c][d]);
            If(matrix[c][d] == 0)
                Counter++;  // same as counter = counter + 1
        }
    }

    // printing the matrix
    Printf("\nThe entered matrix is:\n");
    For(c = 0; c < m; c++)   // to iterate the rows
    {
        For(d = 0; d < n; d++)   // to iterate the columns
        {
            Printf("%d\t", matrix[c][d]);
        }
        Printf("\n"); // to take the control to the next row
    }

    // checking if the matrix is sparse or not
    If(counter > (m * n) / 2)
```

```
   Printf("\nThe entered matrix is a sparse matrix.\n");
Else
   Printf("\nThe entered matrix is not a sparse matrix.\n");

// Adding two matrices (assuming they are sparse)
If(counter > (m * n) / 2)
{
   Int matrix2[10][10];
   Int sum_matrix[10][10];

   Printf("\nEnter the elements of the second matrix:\n");
   For(c = 0; c < m; c++)
   {
      For(d = 0; d < n; d++)
      {
         Scanf("%d", &matrix2[c][d]);
      }
   }

   Printf("\nThe second matrix is:\n");
   For(c = 0; c < m; c++)
   {
      For(d = 0; d < n; d++)
      {
         Printf("%d\t", matrix2[c][d]);
         Sum_matrix[c][d] = matrix[c][d] + matrix2[c][d];
      }
      Printf("\n");
   }

   Printf("\nAddition of the two sparse matrices is:\n");
   For(c = 0; c < m; c++)
   {
      For(d = 0; d < n; d++)
      {
            Printf(“%d\t”, sum_matrix[c][d]);

          }

          Printf(“\n”);

      }

  }

  Else

  {

      Printf(“\nSorry, addition of two normal matrices is not implemented.\n”);

  }


  Printf(“\nCoding is Fun!\n”);

  Return 0;

}
```

Write a c program to Check that the above matrix is sparse or not? If Not, add same Matrix twice and display new matrix. If Yes, print the result as given matrix is a sparse matrix.

1 0 3 4

0 0 2 9

8 6 5 1

0 7 0 4

```
#include <stdio.h>


#define ROWS 4

#define COLS 4


int main() {

  int matrix[ROWS][COLS] = {

      {1, 0, 3, 4},

      {0, 0, 2, 9},

      {8, 6, 5, 1},

      {0, 7, 0, 4}

  };


  int i, j, count = 0;

  int isSparse = 0;
```

```
// Counting the number of zero elements in the matrix
for (i = 0; i < ROWS; i++) {
    for (j = 0; j < COLS; j++) {
        if (matrix[i][j] == 0) {
            count++;
        }
    }
}

// Checking if the matrix is sparse
if (count > (ROWS * COLS) / 2) {
    isSparse = 1;
    printf("The given matrix is a sparse matrix.\n");
}

// If not sparse, adding the matrix to itself
if (!isSparse) {
    int newMatrix[ROWS][COLS];

    // Adding the matrix to itself
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            newMatrix[i][j] = matrix[i][j] + matrix[i][j];
        }
    }

    // Displaying the new matrix
    printf("The new matrix after adding itself:\n");
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            printf("%d ", newMatrix[i][j]);

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

```
    return 0;
}
```

Explain

Sure! Let me explain the program to you step by step:

1. The program starts by including the necessary header file `stdio.h` which provides input and output functions.

2. The program defines the constants `ROWS` and `COLS` to represent the number of rows and columns in the matrix, respectively.

3. In the `main` function, a 2-dimensional array `matrix` is declared and initialized with the given values.

4. Next, the program initializes variables `i`, `j`, `count`, and `isSparse`. `count` is used to keep track of the number of zero elements in the matrix, and `isSparse` is a flag variable to check whether the matrix is sparse or not.

5. The program then uses nested `for` loops to iterate through each element of the matrix. If an element is found to be zero, the `count` variable is incremented.

6. After counting the zero elements, the program checks if the count is greater than half of the total number of elements in the matrix. If it is, then the matrix is considered sparse, and the `isSparse` flag is set to 1. The program also prints a message stating that the matrix is sparse.

7. If the matrix is not sparse (i.e., the `isSparse` flag is 0), the program proceeds to create a new matrix called `newMatrix`.

8. The program uses nested `for` loops to iterate through each element of the original matrix and adds the corresponding element to itself to generate the values for `newMatrix`.

9. Finally, the program displays the new matrix by using nested `for` loops and printing each element.

That's how the program works! It checks if the given matrix is sparse or not, and if it's not sparse, it adds the matrix to itself and displays the new matrix. In the given example, the program will add the matrix to itself and display the new matrix since it is not sparse.

After checking sparse condition for below matrix, represent it in its dense form along-with address. Write a c program to After checking sparse condition for below matrix add two same size Matrices for normal or sparse matrices and display new matrix.

{1 0 2 0 0

2 0 0 0 0

0 3 0 0 0

0 0 0 0 4

0 0 0 2 0}

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

```
#define ROWS 5

#define COLS 5


typedef struct {

    int row;

    int col;

    int value;

} Element;


typedef struct {

    Element data[ROW * COL];

    int rows;

    int cols;

    int count;

} SparseMatrix;


void initializeMatrix(int matrix[ROWS][COLS]) {

    int i, j;

    int elements[ROWS][COLS] = {

        {1, 0, 2, 0, 0},

        {2, 0, 0, 0, 0},

        {0, 3, 0, 0, 0},

        {0, 0, 0, 0, 4},

        {0, 0, 0, 2, 0}

    };


    for (i = 0; i < ROWS; i++) {

        for (j = 0; j < COLS; j++) {

            matrix[i][j] = elements[i][j];

        }

    }

}


void displayMatrix(int matrix[ROWS][COLS]) {
    int i, j;

    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int isSparseMatrix(int matrix[ROWS][COLS]) {
    int i, j;
    int count = 0;

    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            if (matrix[i][j] != 0) {
                count++;
            }
        }
    }

    float density = (float) count / (ROWS * COLS);

    if (density < 0.5) {
        return 1;
    }

    return 0;
}

void addMatrices(int matrix1[ROWS][COLS], int matrix2[ROWS][COLS], int result[ROWS][COLS]) {
    int i, j;

    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }
}

int main() {
    int matrix1[ROWS][COLS];
    int matrix2[ROWS][COLS];
    int result[ROWS][COLS];

    initializeMatrix(matrix1);
    initializeMatrix(matrix2);

    printf("Matrix 1:\n");
    displayMatrix(matrix1);
    printf("\n");

    printf("Matrix 2:\n");
    displayMatrix(matrix2);
    printf("\n");

    if (isSparseMatrix(matrix1) && isSparseMatrix(matrix2)) {
        printf("Both matrices are sparse matrices.\n");
    } else {
        printf("At least one of the matrices is not a sparse matrix.\n");
    }

    addMatrices(matrix1, matrix2, result);

    printf("\nResultant Matrix:\n");
    displayMatrix(result);

    return 0;
}
```

Explain

The given program is a C program that performs the addition of two matrices. It includes functions to initialize a matrix, display a matrix, check if a matrix is sparse, and add two matrices.

Here's an explanation of the program:

1. The program starts by including the necessary header files and defining constants for the number of rows and columns in the matrices.

2. Two structures are defined: `Element` represents a single element in a sparse matrix (with row, column, and value), and `SparseMatrix` represents a sparse matrix (an array of elements along with row and column counts).

3. The `initializeMatrix` function is used to initialize a matrix with predefined values. It copies the elements from a predefined 2D array into the given matrix.

4. The `displayMatrix` function is used to display a matrix. It iterates over the rows and columns of the matrix and prints each element.

5. The `isSparseMatrix` function checks if a given matrix is a sparse matrix. It counts the number of non-zero elements in the matrix and calculates the density (non-zero count divided by the total number of elements). If the density is less than 0.5, it considers the matrix as sparse and returns 1; otherwise, it returns 0.

6. The `addMatrices` function performs the addition of two matrices. It iterates over the rows and columns of the matrices, adds the corresponding elements, and stores the result in the `result` matrix.

7. In the `main` function, three matrices (`matrix1`, `matrix2`, and `result`) are declared.

8. The `initializeMatrix` function is called to initialize `matrix1` and `matrix2` with predefined values.

9. The `displayMatrix` function is called to print the contents of `matrix1` and `matrix2`.

10. The `isSparseMatrix` function is called to check if both `matrix1` and `matrix2` are sparse matrices. If they are both sparse matrices, a message is printed; otherwise, a different message is printed.

11. The `addMatrices` function is called to add `matrix1` and `matrix2`, storing the result in the `result` matrix.

12. The `displayMatrix` function is called to print the `result` matrix.

This program demonstrates how to add two matrices, considers the sparse condition, and provides the resulting matrix as output.

Write a c program to After checking sparse condition for below matrix, represent it in its dense form along-with address.

{0 4 0 5

0 0 3 6

0 0 2 0

2 0 0 0

1 0 0 0}

```
#include <stdio.h>

#define ROWS 5
#define COLS 4

Void sparseToDense(int sparseMatrix[ROWS][COLS], int denseMatrix[ROWS][COLS])
{
   Int I, j;
   Int denseIndex = 0;

   For (I = 0; I < ROWS; i++)
   {
      For (j = 0; j < COLS; j++)
      {
         If (sparseMatrix[i][j] != 0)
         {
            denseMatrix[denseIndex][0] = I;
            denseMatrix[denseIndex][1] = j;
            denseMatrix[denseIndex][2] = sparseMatrix[i][j];
            denseIndex++;
         }
      }
   }
}

Void printMatrix(int matrix[ROWS][COLS])
{
   Int I, j;
```

```
    For (I = 0; I < ROWS; i++)
    {
        For (j = 0; j < COLS; j++)
        {
            Printf("%d ", matrix[i][j]);
        }
        Printf("\n");
    }
}

Int main()
{
    Int sparseMatrix[ROWS][COLS] = {
        {0, 4, 0, 5},
        {0, 0, 3, 6},
        {0, 0, 2, 0},
        {2, 0, 0, 0},
        {1, 0, 0, 0}
    };

    Int denseMatrix[ROWS * COLS][3];
    Int I;

    Printf("Sparse Matrix:\n");
    printMatrix(sparseMatrix);

    sparseToDense(sparseMatrix, denseMatrix);

    printf("\nDense Matrix:\n");
    for (I = 0; I < (ROWS * COLS); i++)
    {
        Printf("%d %d %d\n", denseMatrix[i][0], denseMatrix[i][1], denseMatrix[i][2]);
    }

    Return 0;
```

```
}
```

Explain

This C program takes a sparse matrix as input and converts it into its dense form. Let’s go through the code step by step:

1. The `sparseToDense` function takes two matrix parameters: `sparseMatrix`, which represents the input sparse matrix, and `denseMatrix`, which will store the dense representation of the matrix. It uses nested loops to iterate over each element of the `sparseMatrix`.

2. Inside the loops, it checks if the current element of the `sparseMatrix` is non-zero (`sparseMatrix[i][j] != 0`). If it is non-zero, it means we have a non-zero element in the sparse matrix.

3. In that case, it assigns the row index `i` to `denseMatrix[denseIndex][0]`, the column index `j` to `denseMatrix[denseIndex][1]`, and the value of the non-zero element `sparseMatrix[i][j]` to `denseMatrix[denseIndex][2]`. The `denseIndex` variable keeps track of the current position in the `denseMatrix` array.

4. After processing all the elements of the `sparseMatrix`, the function will have filled the `denseMatrix` with the corresponding non-zero elements and their indices.

5. The `printMatrix` function is a helper function used to print the contents of a matrix. It takes a matrix parameter and uses nested loops to iterate over the rows and columns, printing each element.

6. In the `main` function, a sparse matrix `sparseMatrix` is defined and initialized with the given values.

7. An empty `denseMatrix` is also declared with dimensions `ROWS * COLS` to accommodate the maximum possible non-zero elements.

8. The program first prints the input sparse matrix using the `printMatrix` function.

9. Then, the `sparseToDense` function is called to convert the `sparseMatrix` into its dense form and store the result in `denseMatrix`.

10. Finally, the program prints the dense matrix along with the corresponding row index, column index, and element value using a loop and the `printf` function.

The output of the program displays the input sparse matrix and the resulting dense matrix representation with the respective indices and element values.

Write an algorithm to convert the sparse matrix into the dense matrix. Take any TWO 5X6 matrices, add them and display the difference between regular and sparse matrix in terms of saving of memory space in C

```
#include <stdio.h>

#define ROWS 5

#define COLS 6


// Structure to represent a sparse matrix element

Struct Element {

   Int row;

   Int col;

   Int value;

};


// Function to convert a sparse matrix into a dense matrix

Void convertToDense(int sparseMatrix[][COLS], int denseMatrix[][COLS]) {

   Int I, j, k;

   Struct Element elements[ROWS * COLS];


   // Initialize denseMatrix with 0s

   For (I = 0; I < ROWS; i++) {

      For (j = 0; j < COLS; j++) {

         denseMatrix[i][j] = 0;

      }

   }


   // Convert sparseMatrix to elements array

   K = 0;

   For (I = 0; I < ROWS; i++) {

      For (j = 0; j < COLS; j++) {

         If (sparseMatrix[i][j] != 0) {

            Elements[k].row = I;

            Elements[k].col = j;

            Elements[k].value = sparseMatrix[i][j];
            K++;
          }
        }
    }

    // Fill denseMatrix using elements array
    For (I = 0; I < k; i++) {
        denseMatrix[elements[i].row][elements[i].col] = elements[i].value;
    }
}

// Function to calculate memory space saved
Int calculateMemorySaved(int sparseMatrix[][COLS], int denseMatrix[][COLS]) {
    Int sparseSize = 0;
    Int denseSize = 0;

    // Calculate size of sparse matrix
    Int I, j;
    For (I = 0; I < ROWS; i++) {
        For (j = 0; j < COLS; j++) {
            If (sparseMatrix[i][j] != 0) {
                sparseSize += sizeof(int);
            }
        }
    }

    // Calculate size of dense matrix
    denseSize = ROWS * COLS * sizeof(int);

    return sparseSize – denseSize;
}

// Function to display a matrix
Void displayMatrix(int matrix[][COLS]) {
    Int I, j;
    For (I = 0; I < ROWS; i++) {
        For (j = 0; j < COLS; j++) {
            Printf("%d ", matrix[i][j]);
        }
        Printf("\n");
    }
}

Int main() {
    Int sparseMatrix1[ROWS][COLS] = {
        {0, 0, 3, 0, 4, 0},
        {0, 0, 5, 7, 0, 0},
        {0, 0, 0, 0, 0, 0},
        {0, 2, 6, 0, 0, 0},
        {1, 0, 0, 0, 0, 0}
    };

    Int sparseMatrix2[ROWS][COLS] = {
        {0, 0, 0, 0, 0, 0},
        {0, 1, 2, 0, 0, 0},
        {0, 0, 0, 0, 9, 0},
        {0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 8}
    };

    Int denseMatrix[ROWS][COLS];

    // Convert sparse matrices to dense matrices
    convertToDense(sparseMatrix1, denseMatrix);
    printf("Dense Matrix 1:\n");
    displayMatrix(denseMatrix);
    printf("\n");

    convertToDense(sparseMatrix2, denseMatrix);
    printf("Dense Matrix 2:\n");
    displayMatrix(denseMatrix);

    printf("\n");


    // Calculate memory space saved

    Int memorySaved1 = calculateMemorySaved(sparseMatrix1, denseMatrix);

    Int memorySaved2 = calculateMemorySaved(sparseMatrix2, denseMatrix);


    Printf("Memory space saved for Matrix 1: %d bytes\n", memorySaved1);

    Printf("Memory space saved for Matrix 2: %d bytes\n", memorySaved2);


    Return 0;

}
```

The provided code is an implementation of the algorithm to convert a sparse matrix into a dense matrix and calculate the memory space saved. Let's go through the code step by step:

1. The code includes the necessary headers and defines the number of rows and columns of the matrices using the `#define` directive.

2. A structure named `Element` is defined to represent a non-zero element in the sparse matrix. It contains three members: `row` to store the row index, `col` to store the column index, and `value` to store the value of the element.

3. The `convertToDense` function takes a sparse matrix and a dense matrix as parameters. It first initializes the dense matrix with zeros. Then, it iterates over the sparse matrix and if a non-zero element is found, it stores the element's row, column, and value in the `elements` array of `Element` structures. After that, it fills the dense matrix using the data from the `elements` array.

4. The `calculateMemorySaved` function calculates the memory space saved by the sparse matrix representation compared to the dense matrix representation. It iterates over the sparse matrix, and for each non-zero element, it increments the `sparseSize` variable by the size of an integer. Then, it calculates the size of the dense matrix by multiplying the number of elements (`ROWS * COLS`) with the size of an integer. Finally, it returns the difference between `sparseSize` and `denseSize`.

5. The `displayMatrix` function is used to display a matrix. It iterates over the rows and columns of the matrix and prints each element.

6. In the `main` function, two sparse matrices `sparseMatrix1` and `sparseMatrix2` are defined with some example values. The `denseMatrix` is also defined to store the dense representation of the matrices.

7. The `convertToDense` function is called twice, once for each sparse matrix, to convert them into dense matrices.

8. After converting to dense matrices, the `displayMatrix` function is used to print the dense matrices.

9. The `calculateMemorySaved` function is called to calculate the memory space saved for each sparse matrix compared to the corresponding dense matrix.

10. Finally, the memory space saved for each matrix is printed.

This code allows you to see the dense representation of the provided sparse matrices and calculates the memory space saved. You can modify the sparse matrices or provide your own matrices to observe the results.

Write a C program to convert a sparse matrix to a dense matrix and perform addition on two sparse matrices.

```
#include <stdio.h>

#define MAX_SIZE 100

typedef struct {
   int row;
   int col;
   int value;
} Element;

void convertToDense(Element sparse[], int size, int dense[][MAX_SIZE], int rows, int cols) {
   int i, j;

   for (i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++) {
         dense[i][j] = 0;
      }
   }

   for (i = 0; i < size; i++) {
      dense[sparse[i].row][sparse[i].col] = sparse[i].value;
   }
}

void addSparseMatrices(Element sparse1[], int size1, Element sparse2[], int size2, Element result[], int *size3) {
    int i, j, k;
    i = j = k = 0;

    while (i < size1 && j < size2) {
        if (sparse1[i].row < sparse2[j].row || (sparse1[i].row == sparse2[j].row && sparse1[i].col < sparse2[j].col)) {
            result[k++] = sparse1[i++];
        } else if (sparse1[i].row > sparse2[j].row || (sparse1[i].row == sparse2[j].row && sparse1[i].col > sparse2[j].col)) {
            result[k++] = sparse2[j++];
        } else {
            result[k].row = sparse1[i].row;
            result[k].col = sparse1[i].col;
            result[k++].value = sparse1[i++].value + sparse2[j++].value;
        }
    }

    while (i < size1) {
        result[k++] = sparse1[i++];
    }

    while (j < size2) {
        result[k++] = sparse2[j++];
    }

    *size3 = k;
}

void displaySparseMatrix(Element sparse[], int size) {
    int i;

    printf("Row\tColumn\tValue\n");
    for (i = 0; i < size; i++) {
        printf("%d\t%d\t%d\n", sparse[i].row, sparse[i].col, sparse[i].value);
    }
}
```

```
void displayDenseMatrix(int dense[][MAX_SIZE], int rows, int cols) {
    int i, j;

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

int main() {
    Element sparse1[] = {{0, 0, 1}, {1, 1, 2}, {2, 2, 3}};
    Element sparse2[] = {{0, 0, 4}, {1, 1, 5}, {2, 2, 6}, {2, 3, 7}};
    Element result[MAX_SIZE];
    int size1 = sizeof(sparse1) / sizeof(sparse1[0]);
    int size2 = sizeof(sparse2) / sizeof(sparse2[0]);
    int size3;
    int dense1[MAX_SIZE][MAX_SIZE];
    int dense2[MAX_SIZE][MAX_SIZE];
    int denseResult[MAX_SIZE][MAX_SIZE];

    convertToDense(sparse1, size1, dense1, 3, 3);
    convertToDense(sparse2, size2, dense2, 3, 4);

    printf("Dense Matrix 1:\n");
    displayDenseMatrix(dense1, 3, 3);

    printf("\nDense Matrix 2:\n");
    displayDenseMatrix(dense2, 3, 4);

    addSparseMatrices(sparse1, size1, sparse2, size2, result, &size3);

    printf("\nResultant Sparse Matrix:\n");
    displaySparseMatrix(result, size3);
    return 0;

}
```

Certainly! Let's go through the program step by step to understand what it does:

1. First, the necessary header files are included, and a macro `MAX_SIZE` is defined to represent the maximum size of the matrices.

2. Next, a structure `Element` is defined to represent each element in the sparse matrix. It consists of three fields: `row` to store the row index, `col` to store the column index, and `value` to store the value at that position.

3. The `convertToDense` function takes a sparse matrix represented by an array of `Element` (`sparse[]`) and converts it into a dense matrix represented by a 2D array (`dense[][]`). The `size` parameter indicates the number of elements in the sparse matrix, and `rows` and `cols` specify the dimensions of the dense matrix. This function initializes the dense matrix with zeros and then assigns the corresponding values from the sparse matrix to the dense matrix.

4. The `addSparseMatrices` function performs addition on two sparse matrices represented by `sparse1[]` and `sparse2[]`. It takes the sizes of both matrices (`size1` and `size2`), and stores the result in the `result[]` array. The `size3` parameter is a pointer that stores the size of the resultant sparse matrix. This function iterates over the elements of both sparse matrices, compares their row and column indices, and performs the addition or copies the elements accordingly. The size of the resultant matrix is stored in `size3`.

5. The `displaySparseMatrix` function is used to display a sparse matrix. It takes an array of `Element` (`sparse[]`) and its size as parameters. It iterates over the elements and prints the row, column, and value of each element.

6. The `displayDenseMatrix` function displays a dense matrix. It takes a 2D array (`dense[][]`) representing the matrix and its dimensions (`rows` and `cols`) as parameters. It iterates over the matrix and prints each element.

7. In the `main` function, sample sparse matrices `sparse1` and `sparse2` are defined, along with an array `result` to store the resultant sparse matrix. The sizes of `sparse1` and `sparse2` are calculated using the `sizeof` operator and divided by the size of a single element (`sizeof(sparse1[0])` and `sizeof(sparse2[0])`).

8. Two dense matrices, `dense1` and `dense2`, are declared to convert the sparse matrices to dense form. They have sizes of `MAX_SIZE` rows and columns.

9. The `convertToDense` function is called twice to convert `sparse1` and `sparse2` to dense matrices.

10. The `displayDenseMatrix` function is used to display `dense1` and `dense2`.

11. The `addSparseMatrices` function is called to add `sparse1` and `sparse2`. The result is stored in the `result` array, and the size of the resultant matrix is stored in `size3`.

12. The `displaySparseMatrix` function is used to display the resultant sparse matrix.

13. The program ends with the `return 0;` statement.

This program demonstrates how to convert a sparse matrix to a dense matrix and perform addition on two sparse matrices. The converted dense matrices and the resultant sparse matrix are displayed for verification.

Check that the given matrix is sparse or not. If Yes, add same sparse matrix taking it twice. If Not, print result as Given matrix is not a sparse matrix.

{0100

5009

0030

2040} Write a c program

```
#include <stdio.h>

#define ROWS 4
#define COLS 4

Int main() {
    Int matrix[ROWS][COLS] = {
        {0, 1, 0, 0},
        {5, 0, 0, 9},
        {0, 0, 3, 0},
        {2, 0, 4, 0}
    };

    Int sparse = 1;
    Int I, j;

    // Checking if matrix is sparse
    For (I = 0; I < ROWS; i++) {
        For (j = 0; j < COLS; j++) {
            If (matrix[i][j] != 0) {
                Sparse = 0;
                Break;
            }
        }
        If (!sparse) {
            Break;
        }
    }

    If (sparse) {
        Printf("Given matrix is sparse.\n");

        // Adding the matrix to itself
        Int sparseMatrix[ROWS][COLS];
        For (I = 0; I < ROWS; i++) {
            For (j = 0; j < COLS; j++) {
                sparseMatrix[i][j] = matrix[i][j] + matrix[i][j];
            }
        }

        // Printing the resulting sparse matrix
        Printf("Resulting sparse matrix:\n");
        For (I = 0; I < ROWS; i++) {
            For (j = 0; j < COLS; j++) {
                Printf("%d ", sparseMatrix[i][j]);
            }
            Printf("\n");
        }
    } else {
        Printf("Given matrix is not a sparse matrix.\n");
    }

    Return 0;
}
```

Sure! Let's go through the program step by step:

1. The program starts by including the necessary header file, `stdio.h`, which provides input/output functions.

2. The program defines constants `ROWS` and `COLS` to represent the number of rows and columns in the matrix, respectively.

3. In the `main()` function, a 4x4 matrix `matrix` is declared and initialized with the given values.

4. Two integer variables, `sparse` (initialized as `1`) and loop counters `i` and `j`, are declared.

5. The program then checks if the matrix is sparse by iterating through all its elements using nested `for` loops. If a non-zero value is found, the `sparse` variable is set to `0` and the loops are terminated using `break` statements.

6. After checking all elements, the program examines the `sparse` variable. If it is still `1`, the matrix is considered sparse, and the program proceeds to the next steps. Otherwise, it prints that the given matrix is not sparse and terminates.

7. Inside the conditional block for a sparse matrix, a new matrix `sparseMatrix` is declared to store the result of adding the original matrix to itself.

8. Nested `for` loops are used to iterate through each element of the `matrix`. The corresponding elements from the `matrix` are added together and stored in the `sparseMatrix`.

9. After adding all elements, the resulting `sparseMatrix` is printed row by row using nested `for` loops and `printf()`.

10. Finally, the program ends, and the `main()` function returns `0`.

The program determines if the given matrix is sparse by checking if any non-zero elements exist. If it is sparse, it adds the matrix to itself and prints the resulting matrix. Otherwise, it simply states that the given matrix is not sparse.