UNIT – 2

Arrays:

An array is a collection of variables of the same data type stored at contiguous memory locations. Arrays allow us to store multiple values in a single variable, which can be accessed using indices. Arrays in C can be either one-dimensional (1D) or multi-dimensional (e.g., two-dimensional).

1. One-Dimensional Arrays (1D Arrays)

A one-dimensional array is a linear list of elements that are all of the same type. The size of an array must be declared when it is created.

Declaring a 1D Array

Syntax:

```
data_type array_name[size];
```

- data_type: Type of elements (e.g., int, float, char, etc.).
- array_name: Name of the array.
- size: Number of elements in the array.

Example:

```
int numbers[5];
```

- This creates an integer array numbers with 5 elements.

Initializing a 1D Array: Arrays can be initialized at the time of declaration.

Example:

```
int numbers[5] = {10, 20, 30, 40, 50};
```

- If the size is omitted, it is automatically determined by the number of elements in the initializer.
- int numbers[] = {10, 20, 30, 40, 50}; // Size is automatically 5

Accessing and Manipulating Elements: Array elements are accessed using the index operator []. The index starts from 0 and goes up to size-1.

Example:

```
numbers[0] = 100; // Changes the first element to 100

int x = numbers[2]; // Accesses the third element (index 2)
```

Can loop through an array:

Example:

```
for(int i = 0; i < 5; i++)

{

    printf("%d ", numbers[i]);

}
```

2. Two-Dimensional Arrays (2D Arrays)

A two-dimensional array is essentially an array of arrays. It can be visualized as a table with rows and columns.

Declaring a 2D Array:

Syntax:

```
data_type array_name[rows][columns];
```

Example:

```
int matrix[3][4]; // 3 rows, 4 columns
```

Initializing a 2D Array:

Example:

```
int matrix[3][4] =

{

 {1, 2, 3, 4},

{5, 6, 7, 8},

 {9, 10, 11, 12}

};
```

Accessing and Manipulating Elements:

Elements in a 2D array are accessed using two indices: one for the row and one for the column.

Example:

matrix[0][2] = 10; // Changes the value in the first row, third column to 10

int y = matrix[2][1]; // Accesses the third row, second column

Can loop through all elements:

Example:

```
for(int i = 0; i < 3; i++)
 {
  for(int j = 0; j < 4; j++)
 {
        printf("%d ", matrix[i][j]);
 }
  printf("\n");
}
```

Base Address of an Array:

The base address of an array is simply the address of the first element (arr[0]). In C, this can be retrieved as:

- &arr[0] or simply arr (since the array name itself gives the base address).

a) 1D Array Address Calculation

For a 1D array arr[] of size n, the address of the element arr[i] is calculated as:

Address of arr[i]=Base Address of array+i×Size of each element

Where:

- Base Address is the address of the first element arr[0].
- i is the index of the element you want to access.
- Size of each element is the size of the data type used in the array (e.g., 4 bytes for int on most systems).

b) 2D Array Address Calculation

For a 2D array arr[m][n], the address of the element at row i and column j is calculated as:

Address of arr[i][j]=Base Address of array+[(i×Number of columns)+j]×Size of each element

Where:

- i is the row index.
- j is the column index.
- Number of columns is the number of columns in each row (second dimension).
- Size of each element is the size of the data type used in the array.

Reading and writing of elements (1-D array):

```
#include <stdio.h>

int main()

{

    int array[100], n, i;

    // Input the size of the array

    printf("Enter the number of elements in the array: ");

    scanf("%d", &n);

    // Reading elements into the array

    printf("Enter the elements of the array:\n");

    for (i = 0; i < n; i++)

   {

        scanf("%d", &array[i]);

    }

    // Displaying the elements of the array

    printf("The elements of the array are:\n");

    for (i = 0; i < n; i++)

  {

       printf("Element [%d]: %d\n", i, array[i]);

    }

    return 0;

}
```

Output:

```
Enter the number of elements in the array: 5

Enter the elements of the array:

10

20

30

40

50

The elements of the array are:

Element [0]: 10

Element [1]: 20

Element [2]: 30

Element [3]: 40

Element [4]: 50
```

Reading and writing of elements (2-D array):

```
#include <stdio.h>

int main() {

    int rows, cols;

    // Taking input for the size of the 2D array (rows and columns)

    printf("Enter the number of rows: ");

    scanf("%d", &rows);

    printf("Enter the number of columns: ");

    scanf("%d", &cols);

    int matrix[rows][cols];  // Declare the 2D array

    // Reading elements into the 2D array
    printf("\nEnter elements of the matrix:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("Enter element at matrix[%d][%d]: ", i, j);
            scanf("%d", &matrix[i][j]);
        }
    }
    // Writing (displaying) the elements of the 2D array
    printf("\nMatrix elements are:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");  // Newline after each row
    }
    return 0;
}
```

Output:

```
Enter the number of rows: 2
Enter the number of columns: 3
Enter elements of the matrix:
Enter element at matrix[0][0]: 1
Enter element at matrix[0][1]: 2
Enter element at matrix[0][2]: 3
Enter element at matrix[1][0]: 4
Enter element at matrix[1][1]: 5

Enter element at matrix[1][2]: 6

Matrix elements are:

1 2 3

4 5 6
```

3a) Write a C program to find the minimum, maximum in an array of integers.

Algorithm:

```
1. Start.
2. Declare an integer array arr[100], and variables i, max, min, and size.
3. Input the size of the array:
    • Prompt the user to enter the size of the array and store it in size.
4. Input the elements of the array:
    • Use a loop from i = 0 to i = size-1.
    • In each iteration, prompt the user to enter the element, and store it in arr[i].
5. Initialize the maximum (max) and minimum (min) values with the first element of the array (arr[0]).
6. Loop through the array starting from the second element (i = 1) to find the maximum and minimum values:
    • For each element arr[i]:
        o If arr[i] > max, update max = arr[i].
        o If arr[i] < min, update min = arr[i].
7. Output the values of max and min:
    • Print the maximum value.
    • Print the minimum value.
8. End.
```

//C program

```c
#include <stdio.h>
int main()
{
int arr[100];
int i, max, min, size;
printf("Enter size of the array: ");
scanf("%d", &size);
printf("Enter elements in the array: ");
for(i=0; i<size; i++)
{
scanf("%d", &arr[i]);
}
max = arr[0];
min = arr[0];
for(i=1; i<size; i++)
{
if(arr[i] > max)
{
max = arr[i];
}
if(arr[i] < min)
{
min = arr[i];
}
}
printf("Maximum element = %d\n", max);
printf("Minimum element = %d", min);
return 0;
}
```

Output:

```
Enter size of the array: 3
Enter elements in the array: 4
0
8
Maximum element = 8
Minimum element = 0
```

3b) Write a C to compute mean, variance, Standard Deviation of n elements in a single dimension array.

Algorithm:

1. Read the number of elements and the elements into the array.

2. Compute Mean:

- Calculate the sum of all elements.
- Divide the sum by the number of elements to get the mean.

$$\text{Mean}(\mu) = \frac{\sum_{i=1}^{n} x_i}{n}$$

3. Compute Variance:

- Calculate the squared differences between each element and the mean.
- Sum these squared differences.
- Divide the sum by the number of elements to get the variance.

$$\text{Variance}(\sigma^2) = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$

4. Compute Standard Deviation:

- Take the square root of the variance.

$$\text{Standard Deviation}(\sigma) = \sqrt{\text{Variance}}$$

5. Output Results:

- Print the mean, variance, standard deviation.

//C Program

```
#include <stdio.h>
#include <math.h>
int main()
{
   int arr[100], n, i, j;
   double sum=0.0, sum_sq_diff = 0.0, mean, variance, std_dev;
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   printf("Enter the elements:\n");
   for(i = 0; i < n; i++)
  {
      scanf("%d", &arr[i]);
   }
   // Compute mean
   for(i = 0; i < n; i++)
   {
      sum += arr[i];
```

```
    }
    mean = sum / n;
    printf("Mean: %.2f\n", mean);
    // Compute variance
    for(i = 0; i < n; i++)
   {
       sum_sq_diff += (arr[i] - mean) * (arr[i] - mean);
    }
    variance = sum_sq_diff / n;
     printf("Variance: %.2f\n", variance);
    // Compute standard deviation
    std_dev = sqrt(variance);
    printf("Standard Deviation: %.2f\n", std_dev);
    return 0;
}
```

Output:

```
Enter the number of elements: 5
Enter the elements:
1
3
5
2
1
Mean: 2.40
Variance: 2.24
Standard Deviation: 1.50
```

Q) Write a C program for Addition of Two Matrices

Algorithm:

1.Enter the number of rows (m) and columns (n) of the matrices.

2. Input First Matrix:

- Initialize a 2D array first with dimensions m x n.
- Loop through each element of the first matrix and prompt the user to enter the values. Store these values in the first matrix.

3. Input Second Matrix:

- Initialize a 2D array second with dimensions m x n.
- Loop through each element of the second matrix and prompt the user to enter the values. Store these values in the second matrix.

4.Add Matrices:

- Initialize a 2D array sum with dimensions m x n to store the result.
- Loop through each element of the matrices.
    - For each element, compute the sum of the corresponding elements from the first and second matrices.
    - Store the result in the sum matrix.

5.Display Result:

- Print the sum matrix. Display each element in a tabular format.

// **C Program**

```
#include <stdio.h>
int main()
{
int m, n, c, d, first[10][10], second[10][10], sum[10][10];
printf("Enter the number of rows and columns of matrix\n");
scanf("%d%d", &m, &n);
printf("Enter the elements of first matrix\n");
for (c = 0; c < m; c++)
for (d = 0; d < n; d++)
scanf("%d", &first[c][d]);
printf("Enter the elements of second matrix\n");
for (c = 0; c < m; c++)
for (d = 0 ; d < n; d++)
scanf("%d", &second[c][d]);
printf("Sum of entered matrices:\n");
for (c = 0; c < m; c++)
{
for (d = 0 ; d < n; d++)
{
sum[c][d] = first[c][d] + second[c][d];
printf("%d\t", sum[c][d]);
}
printf("\n");
}
return 0;
}
```

Output:

```
Enter the number of rows and columns of matrix
2
2
Enter the elements of first matrix
3
4
5
6
Enter the elements of second matrix
1
2
3
4
Sum of entered matrices:
4	6
8	10
```

Q) Write a C program for Multiplication of Two Matrices

Algorithm:

1. Input Matrix Dimensions:
   - Enter number of rows (m) and columns (n) for the first matrix.
   - Enter number of rows (p) and columns (q) for the second matrix.
2. Check Multiplication Condition:
   - Check if the number of columns in the first matrix (n) is equal to the number of rows in the second matrix (p).
     - If n != p, print an error message and stop the program, as matrix multiplication cannot be performed in this case.
3. Input First Matrix:
   - Use a nested loop to read elements into the first matrix with dimensions m x n.
4. Input Second Matrix:
   - Use a nested loop to read elements into the second matrix with dimensions p x q.

5. Multiply the Matrices:

- Initialize a result matrix multiply with dimensions m x q to store the product.
- For each element in the result matrix, compute the sum of the product of corresponding row elements from the first matrix and column elements from the second matrix:
    - Loop through each row c in the first matrix.
    - Loop through each column d in the second matrix.
    - For each element at position [c][d] in the product matrix, compute the sum of the products of the corresponding elements from the c-th row of the first matrix and the d-th column of the second matrix.
    - Store the computed sum in the product matrix at position [c][d].

6. Display the Product Matrix:

- Use nested loops to display the resulting product matrix in a tabular format.

//C Program

```
#include <stdio.h>
int main()
{
int m, n, p, q, c, d, k, sum = 0;
int first[10][10], second[10][10], multiply[10][10];
printf("Enter number of rows and columns of first matrix\n");
scanf("%d%d", &m, &n);
printf("Enter number of rows and columns of second matrix\n");
scanf("%d%d", &p, &q);
if (n != p)
printf("The matrices can't be multiplied with each other.\n");
else
{
printf("Enter elements of first matrix\n");
for (c = 0; c < m; c++)
for (d = 0; d < n; d++)
scanf("%d", &first[c][d]);
printf("Enter elements of second matrix\n");
for (c = 0; c < p; c++)
for (d = 0; d < q; d++)
scanf("%d", &second[c][d]);
for (c = 0; c < m; c++) {
for (d = 0; d < q; d++) {
for (k = 0; k < p; k++) {
```

```
sum = sum + first[c][k]*second[k][d];
}
multiply[c][d] = sum;
sum = 0;
}
}
printf("Product of the matrices:\n");
for (c = 0; c < m; c++) {
for (d = 0; d < q; d++)
printf("%d\t", multiply[c][d]);
printf("\n");
}
}
return 0;
}
```

Output:

Case 1:

```
Enter number of rows and columns of first matrix
2 3
Enter elements of first matrix
1 2 3 4 5 6
Enter number of rows and columns of second matrix
3 2
Enter elements of second matrix
1 2 3 4 5 6
Product of the matrices:
22	28
49	64
```

Case 2:

```
Enter number of rows and columns of first matrix
2 3
Enter number of rows and columns of second matrix
2 3
The matrices can't be multiplied with each other.
```

3D Araay:

It is used to store data in a three-dimensional space, and it extends the concept of a 2D array. The elements of a 3D array are accessed using three indices: one for the depth, one for the rows, and one for the columns.

```
data_type array_name[size1][size2][size3];
```

Where:

- data_type is the type of elements (e.g., int, float, etc.)
- size1 is the number of "layers" (or depth).
- size2 is the number of rows.
- size3 is the number of columns.

Example:

```
int arr[2][3][4];
```

Initialization of a 3D Array:

```
int arr[2][3][4] = {
    {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    },
    {
        {13, 14, 15, 16},
        {17, 18, 19, 20},
        {21, 22, 23, 24}
    }
};
```

Accessing Elements in a 3D Array:

```
int value = arr[0][1][3];  // This refers to 8 in the array.
```

Reading and Displaying:

```
#include <stdio.h>
int main() {
    int arr[2][2][3];  // Declare a 3D array with 2 layers, 2 rows, and 3 columns
    int i, j, k;
    // Reading elements into the 3D array
    printf("Enter the elements of the 3D array:\n");
    for(i = 0; i < 2; i++) {
        for(j = 0; j < 2; j++) {
            for(k = 0; k < 3; k++) {
                printf("Enter element at [%d][%d][%d]: ", i, j, k);
                scanf("%d", &arr[i][j][k]);
            }
        }
    }
    // Displaying the elements of the 3D array
    printf("\nThe elements of the 3D array are:\n");
    for(i = 0; i < 2; i++) {
        for(j = 0; j < 2; j++) {
            for(k = 0; k < 3; k++) {
                printf("Element at [%d][%d][%d] = %d\n", i, j, k, arr[i][j][k]);
            }
        }
    }
    return 0;
}
```

Output:

```
Enter the elements of the 3D array:
Enter element at [0][0][0]: 1
Enter element at [0][0][1]: 2
Enter element at [0][0][2]: 3
Enter element at [0][1][0]: 4
Enter element at [0][1][1]: 5
Enter element at [0][1][2]: 6
Enter element at [1][0][0]: 7
Enter element at [1][0][1]: 8
Enter element at [1][0][2]: 9
Enter element at [1][1][0]: 10
Enter element at [1][1][1]: 11
Enter element at [1][1][2]: 12

The elements of the 3D array are:
Element at [0][0][0] = 1
Element at [0][0][1] = 2
Element at [0][0][2] = 3
Element at [0][1][0] = 4
Element at [0][1][1] = 5
Element at [0][1][2] = 6
Element at [1][0][0] = 7
Element at [1][0][1] = 8
Element at [1][0][2] = 9
Element at [1][1][0] = 10
Element at [1][1][1] = 11
Element at [1][1][2] = 12
```

Strings:

- A string is an array of characters terminated by a special character '\0' (null character).
- Strings can be declared, initialized, and manipulated using standard C functions from the string.h library.
- A string is stored as a sequence of characters in memory, and the end of the string is marked by the null terminator.

Fixed-Length Strings

- A fixed-length string has a predefined size and cannot store more characters than the size allows. Extra spaces in the array remain unused.
- Example:

```
char fixed_str[10];  // Can store up to 9 characters + 1 null terminator
```

Variable Length Strings

- Variable-length strings are dynamic and may change size during runtime.
- C does not inherently support variable-length strings, but such behavior can be achieved using pointers and dynamic memory allocation (malloc(), realloc()).

  Example:

  ```
  char *str = (char *)malloc(10 * sizeof(char));
  ```

Length-Controlled Strings

- These strings manage their length through an integer variable that stores the number of characters currently in use, excluding the null terminator.

Storing Strings

- Strings are stored in an array of characters. The size of the array determines how many characters can be stored.
- The last character in a string is always the null terminator '\0', which marks the end of the string.

  ```
  char str[] = "Hello";
  ```

- A string literal is a sequence of characters enclosed in double quotes.

  ```
  char *str = "Hello, World!";   // Hello, World! is a string literal.
  ```

- Each character in the string can be accessed by index.

  ```
  char str[] = "Hello";
  char firstChar = str[0]; // 'H'
  ```

String Delimiters

- A delimiter is a character or a sequence of characters that specifies the boundary between separate entities (e.g., commas, spaces, etc.).
- In C, the null character '\0' is used as a delimiter to mark the end of a string.

Declaring Strings

- Strings are declared as arrays of characters.
- Either specify the size of the array or allow the compiler to determine the size based on the string literal.

  Example:

  - char str[10] = "Hello";
  - char str[] = "Hello";

Initializing Strings

- Strings can be initialized at the time of declaration.

- If you initialize a string with a literal, the null character ('\0') is automatically added.
    - Example:
        - str[] = {'H', 'e', 'l', 'l', 'o', '\0'};
        - char str[] = "Hello, World!";
        - char *str = "Hello, World!";

Reading and Writing Strings

- Strings can be read using scanf(), and written using printf().
- When using scanf(), the %s format specifier reads a string, but stops at the first whitespace character.

  Example:

  ```
  char str[100];
  scanf("%s", str); // Reads until whitespace
  printf("%s", str); // Prints string
  ```

String Input/Output Functions

- gets(): Reads a string from standard input until a newline character.
    - Deprecated, use fgets() instead for safety.
- puts(): Writes a string to standard output and adds a newline.
    - str: A pointer to a null-terminated string to be printed.
    - **Return value**: puts() returns a non-negative value if successful and EOF if an error occurs.

Example:

```
char str[100];
fgets(str, 100, stdin); // Safe alternative to gets()
puts(str); // Output string
```

NOTE: gets() reads input from the standard input (stdin) until it encounters a newline character (\n) but does not limit the number of characters read. This means if a user provides more input than the allocated space for the string, it can overwrite adjacent memory, leading to buffer overflow.

Strings as Array of Characters:

- Can iterate through strings and access or manipulate individual characters.

Example:

```
char str[] = "Hello";
for (int i = 0; str[i] != '\0'; i++)
```

```
{
        printf("%c", str[i]); // Access each character
}
```

String Manipulation Functions:

- C standard library provides several functions to manipulate strings in string.h:
- **strlen()**: Returns the length of a string (excluding the null character).

  ```
  int len = strlen(str);
  ```
- **strcat()**: Concatenates two strings.

  ```
  strcat(dest, src); // Appends src to dest
  ```
- **strcpy()**: Copies one string to another.

  ```
  strcpy(dest, src); // Copies src into dest
  ```
- **strcmp()**: Compares two strings lexicographically.

  ```
  int cmp = strcmp(str1, str2); // Returns 0 if strings are equal
  ```
- **strchr()**: Finds the first occurrence of a character in a string.

  ```
  char *pos = strchr(str, 'a'); // Finds 'a' in str
  ```
- **strstr()**: Finds the first occurrence of a substring in a string.

  ```
  char *pos = strstr(str, "world"); // Finds "world" in str
  ```
- **strncpy()**: Copies a certain number of characters from one string to another.
- **strncat()**: Concatenates a certain number of characters from one string to another.
- **strtok()**: Tokenizes a string based on a given delimiter.

//Demonstration of few string functions

```
#include <stdio.h>
#include <string.h>
int main() {
   char str1[100], str2[100], str3[100];
   // Demonstrating strlen()
   printf("Enter a string: ");
   fgets(str1, sizeof(str1), stdin);
```

```
str1[strcspn(str1, "\n")] = 0;  // Remove newline character from fgets

printf("Length of the string '%s' is: %zu\n", str1, strlen(str1));

// Demonstrating strcpy()
strcpy(str2, str1);
printf("String copied to str2: %s\n", str2);

// Demonstrating strcat()
printf("Enter another string to concatenate: ");
fgets(str3, sizeof(str3), stdin);
str3[strcspn(str3, "\n")] = 0;  // Remove newline character from fgets

strcat(str1, str3);
printf("After concatenation, str1 is: %s\n", str1);

// Demonstrating strcmp()
int comparison = strcmp(str1, str2);
if (comparison == 0) {
   printf("str1 and str2 are equal.\n");
} else if (comparison < 0) {
   printf("str1 is less than str2.\n");
} else {
   printf("str1 is greater than str2.\n");
}

// Demonstrating strstr()
char *substring = strstr(str1, str2);
if (substring != NULL) {
   printf("'%s' found in str1.\n", str2);
```

```
    } else {
        printf("'%s' not found in str1.\n", str2);
    }
    return 0;
}
```

Arrays of Strings:

- An array of strings is a two-dimensional array where each element is a string.

  Example:

  ```
  char names[5][10] = {"Alice", "Bob", "Charlie", "David", "Eve"};
  for (int i = 0; i < 5; i++)
  {
   printf("%s\n", names[i]); // Prints each name
  }
  ```

a) Write a C program to convert a Roman numeral ranging from I to L to its decimal equivalent.

Algorithm:

1. Start:
   - Initialize a variable decimalValue to 0, which will hold the result.

2.Read Input:
   - Input the Roman numeral string from the user and store it in a character array roman.

3. Determine Length:
   - Calculate the length of the Roman numeral string.

4. Process Each Character:
   - Loop through each character in the Roman numeral string:
     - Step 4.1:
       - Determine the decimal value of the current Roman numeral character using a switch statement:
         - I = 1
         - V = 5
         - X = 10

```
                ▪ L = 50
        o Step 4.2:
            ▪ Determine the decimal value of the next Roman numeral character if it
              exists; otherwise, set it to 0.
        o Step 4.3:
            ▪ Compare the current character's value with the next character's value:
                ▪ If the current value is less than the next value:
                    ▪ Subtract the current value from decimalValue.
                ▪ Otherwise:
                    ▪ Add the current value to decimalValue.
5. Output the Result:
    • Print the final value of decimalValue.
6. End.
```

//C Program

```c
#include <stdio.h>
#include <string.h>
int main()
{
  char roman[20];
  int length, i;
  int decimalValue = 0;
  // Input the Roman numeral from the user
  printf("Enter a Roman numeral (I to L): ");
  scanf("%s", roman);
  length = strlen(roman);
  for (i = 0; i < length; i++) {
    // Get the value of the current Roman numeral character
    int currentVal;
    switch (roman[i])
    {
      case 'I':
            currentVal = 1;
            break;
      case 'V':
            currentVal = 5;
             break;
      case 'X':
            currentVal = 10;
            break;
      case 'L':
            currentVal = 50;
            break;
      default:
            currentVal = 0;
            break;
    }
    // Get the value of the next Roman numeral character (if available)
    int nextVal = (i + 1 < length) ? (roman[i + 1] == 'I' ? 1 :
                                  roman[i + 1] == 'V' ? 5 :
                                  roman[i + 1] == 'X' ? 10 :
                                  roman[i + 1] == 'L' ? 50 : 0) : 0;

    // If the current value is less than the next value, subtract it; otherwise, add it
    if (currentVal < nextVal) {
      decimalValue -= currentVal;
    } else {
      decimalValue += currentVal;
    }
  }
  // Display the decimal equivalent
```

```
    printf("The decimal equivalent is: %d\n", decimalValue);
    return 0;
}
```

f) Write a C program to determine if the given string is a palindrome or not (Spelled same in both directions with or without a meaning like madam, civic, noon, abcba, etc.)

Algorithm:

1.Initialize Variables:

- Create a variable left to point to the beginning of the string.
- Create a variable right to point to the end of the string.

2. Check Palindrome:

- Compare characters from both ends of the string moving towards the center:
    - If the character at the left index is not equal to the character at the right index, the string is not a palindrome.
    - If all characters match, the string is a palindrome.

3. Output Result:

- Print whether the string is a palindrome or not.

//C Program

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int main() {
    char str[100];
    int left, right;
    int isPalindrome = 1;  // Assume the string is a palindrome initially
    // Input the string from the user
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    // Remove newline character if present
    str[strcspn(str, "\n")] = 0;
    // Convert the string to lowercase to handle case-insensitivity
    for (int i = 0; str[i]; i++) {
        str[i] = tolower(str[i]);
    }
    // Initialize pointers
    left = 0;
    right = strlen(str) - 1;
    // Check if the string is a palindrome
    while (left < right) {
        if (str[left] != str[right]) {
            isPalindrome = 0; // Not a palindrome
            break;
        }
        left++;
        right--;
    }
    // Output result
    if (isPalindrome) {
        printf("The string is a palindrome.\n");
    } else {
        printf("The string is not a palindrome.\n");
    }
    return 0;
}
```

g) Write a C program that displays the position of a character ch in the string S or – 1 if S doesn‘t contain ch.

Algorithm:

1. Input Handling:
    - Read the string S from the user.
    - Read the character ch whose position you want to find.

2. Search for Character:
   - Traverse the string S from the beginning to the end.
   - Compare each character with ch.
3. Determine Position:
   - If you find a match, record the position (index) and stop searching.
   - If the end of the string is reached without finding ch, set the position to -1.
4. Output Result:
   - Print the position of the character ch in the string S, or -1 if it is not found.

// C Program

```
#include <stdio.h>
int main() {
   char str[100];
   char ch;
   int i;
   int position = -1;
   // Input the string from the user
   printf("Enter a string: ");
   fgets(str, sizeof(str), stdin);
   // Remove newline character if present
   str[strcspn(str, "\n")] = 0;
   // Input the character to search
   printf("Enter a character to find: ");
   scanf(" %c", &ch);
   // Search for the character in the string
   for (i = 0; str[i] != '\0'; i++) {
      if (str[i] == ch) {
         position = i; // Record the position of the character
         break; // Exit the loop once the character is found
      }
```

```
    }
    // Output the result
    printf("The position of character '%c' is: %d\n", ch, position);
    return 0;
}
```

g) Write a C program to count the lines, words and characters in a given text.

Algorithm:

1.Initialize Counters:

- Set up counters for the number of characters (l), words (f), and lines (lineCount).
- Initialize l and lineCount to 0 and f to 1 (assuming there is at least one word).

2.Read Input:

- Use gets() to read the input string into the str array. (Note: gets() is deprecated and unsafe; in practice, use fgets().)

3.Count Characters and Lines:

- Traverse each character in the string:
  - Increment the character count l for each character read.
  - If a newline character ('\n') is encountered, increment the line count lineCount by 1.

4.Count Words:

- Traverse the string again:
  - Each time a space character (' ') is encountered, increment the word count f by 1.
  - This assumes words are separated by spaces and does not handle multiple spaces or punctuation marks.

5.Output Results:

- Print the total number of characters, words, and lines in the string.

//C Program

```
#include <stdio.h>
#include <string.h>
void main() {
    char str[100];
    int i = 0, l = 0, f = 1, lineCount = 1; // Added lineCount to track number of lines
    puts("Enter any string");
    gets(str); // Note: gets() is unsafe, consider using fgets() in practice
    // Count characters and lines
    for (i = 0; str[i] != '\0'; i++) {
        l = l + 1;
        if (str[i] == '\n') {
            lineCount++;
        }
    }
    // Count words
    for (i = 0; i <= l - 1; i++) {
        if (str[i] == ' ') {
            f = f + 1;
        }
    }
    // Output results
    printf("The number of characters in the string are %d\n", l);
    printf("The number of words in the string are %d\n", f);
    printf("The number of lines in the string are %d\n", lineCount);
}
```

Enumerated Types:

An enumerated type (or enum) in C is a user-defined data type that consists of a set of named integer constants. The enum data type allows you to define variables that can only take one of the predefined values (constants), which makes your code more readable and easier to maintain.

Defining an Enumerated Type:

An **enumeration** is defined using the enum keyword.

Syntax:

enum enum_name {const1, const2, ..., constN};

- enum_name is the name of the enumeration.
- const1, const2, ..., constN are the enumerator constants, which are typically symbolic names representing integer values.

Example:

enum Weekday {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

In this example, an enumeration named Weekday is defined, and its constants are Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday. By default, the constants take integer values starting from 0. So, Sunday is 0, Monday is 1, and so on.

Example:

```
#include <stdio.h>

// Define an enum to represent the days of the week

enum Weekday {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

int main()

{

  // Declare a variable of type enum Weekday

  enum Weekday today;

  // Assign a value from the enum

  today = Wednesday;

  // Use a switch statement to print the name of the day

  switch (today)

{

    case Sunday:

      printf("It's Sunday!\n");

      break;

    case Monday:

      printf("It's Monday!\n");

      break;

    case Tuesday:

      printf("It's Tuesday!\n");

      break;

    case Wednesday:
      printf("It's Wednesday!\n");
      break;
    case Thursday:
      printf("It's Thursday!\n");
      break;
    case Friday:
      printf("It's Friday!\n");
      break;
    case Saturday:
      printf("It's Saturday!\n");
      break;
    default:
      printf("Invalid day!\n");
  }
  return 0;
}
```

Structures:

- A **structure** in C is a user-defined data type that allows combining data items of different kinds.
- Each data item within a structure is called a **member**.
- Structures help in organizing complex data and are essential for handling multiple related variables efficiently.

Defining a Structure

To define a structure, the struct keyword is used followed by the structure name and the members enclosed within curly braces {}. A structure definition does not allocate memory; it merely defines the data layout.

Syntax:

```
struct structure_name {
  data_type member1;
  data_type member2;
  ...
};
```

Example:

```
struct Person {
   char name[50];
   int age;
   float height;
};
```

Declaring Structure Variables:

Once a structure is defined, you can declare variables of that type.

Syntax:

```
struct structure_name variable_name;
```

Example:

```
struct Person person1;
```

Here, person1 is a variable of type struct Person. This variable can store information for one person.

Accessing Structure Members:

To access or assign values to a structure's members, the **dot (.) operator** is used.

Syntax:

```
variable_name.member_name;
```

Example:

```
person1.age = 25;
strcpy(person1.name, "John Doe");  // Using strcpy to assign string
person1.height = 5.9;
```

Initializing Structures:

You can initialize a structure at the time of declaration.

Syntax:

```
struct structure_name variable_name = {value1, value2, ...};
```

Example:

```
struct Person person2 = {"Alice", 30, 5.5};
```

In this example, person2 is initialized with a name "Alice", an age of 30, and a height of 5.5.

Example:

```
struct Person {
   char name[50];
   int age;
   float height;
};
struct Person person1 = {"John Doe", 25, 5.9};
```

Nested Structures:

Structures can also be nested, which means one structure can be a member of another structure.

Example:

```
struct Address {
   char city[30];
   int zip;
};

struct Employee {
   char name[50];
   struct Address emp_address;
};
```

Here, the Employee structure has a member emp_address of type struct Address. Accessing the nested structure is done using multiple dot operators.

```
struct Address {
   char city[30];
   int zip;
};

struct Employee {
   char name[50];
   struct Address emp_address;
};
```

```
struct Employee emp1 = {"John", {"New York", 10001}};
```

Accessing nested members:

```
struct Employee emp1;

emp1.emp_address.zip = 12345;
```

Array of Structures:

You can create an array of structures to store multiple records of the same type.

Syntax:

```
struct structure_name array_name[size];
```

Example:

```
struct Person people[3] = {
    {"John", 25, 5.9},
    {"Alice", 30, 5.5},
    {"Bob", 22, 5.8}
};
```

You can access and modify each structure element using an index, for example:

```
people[0].age = 26;
```

Pointers to Structures:

Like other data types, pointers can be used with structures.

Syntax:

```
struct structure_name *pointer_name;
```

Example:

```
struct Person *ptr;

ptr = &person1;
```

You can access structure members using a pointer and the arrow (->) operator.

Example:

```
ptr->age = 28;
```

Self-Referential Structures:

A structure can contain a pointer to itself, which is useful in creating complex data structures like linked lists.

Example:

```
struct Node {
    int data;
    struct Node *next;
};
```

Typedef with Structures:

You can use the typedef keyword to create an alias for a structure, simplifying its usage.

Syntax:

```
typedef struct {
    data_type member1;
    data_type member2;
} alias_name;
```

Example:

```
typedef struct {
    char name[50];
    int age;
} Person;

Person person3;
```

In this case, the Person alias can be used directly to declare structure variables without the struct keyword.

Size of Structures

The size of a structure is the sum of the sizes of its individual members, but it may also include padding for alignment. The sizeof operator can be used to find the size of a structure.

Example:

```
printf("Size of Person structure: %lu\n", sizeof(struct Person));
```

//C Program

```
#include <stdio.h>
// Define a structure to represent a student
struct Student {
    int rollNumber;
    char name[50];
};

int main() {
    // Declare a variable of type struct Student
    struct Student s1;

    printf("Enter roll number: ");
    scanf("%d", &s1.rollNumber);

    printf("Enter name: ");
    scanf("%s", s1.name);

    // Output the details of the student
    printf("\nStudent Details:\n");
    printf("Roll Number: %d\n", s1.rollNumber);
    printf("Name: %s\n", s1.name);
    return 0;
}
```

Unions:

A **union** is a user-defined data type in C that, like a structure, can hold variables of different data types. However, unlike a structure where each member has its own memory location, a union uses a single shared memory space for all its members. This means that at any given time, a union can store only one of its members, and all the members share the same memory location.

Syntax:

```
union union_name {
   data_type1 member1;
   data_type2 member2;
   ...
};
```

Example:

```
union Data {
    int i;
    float f;
    char str[20];
};
```

Example:

```
#include <stdio.h>

union Data {
    int i;
    float f;
    char str[20];
};

int main()
 {
    union Data data;
    data.i = 10;
    printf("data.i : %d\n", data.i);
    data.f = 220.5;
    printf("data.f : %f\n", data.f);
    return 0;
}
```

Difference between Structure and Union:

	Structure	**Union**
Memory Allocation	Allocates separate memory for each member.	All members share the same memory location.
Size	Total size is the sum of all members' sizes.	Size is equal to the largest member's size.

Member Accessibility	All members can store values simultaneously.	Only one member can store a value at a time.
Memory Usage	Requires more memory since all members are stored separately.	More memory-efficient as only one member occupies memory.
Member Overwriting	No member overwrites another's value.	Assigning a value to one member overwrites other members' values.
Use Case	Useful when storing multiple values at the same time.	Useful when storing one of many values, optimizing memory usage.
Example Use	Struct for storing multiple related pieces of data, e.g., student information.	Union for saving memory when handling multiple data types but only using one at a time.
Accessing Members	Can access and modify multiple members independently.	Can access and modify only one member at a time.

Pointers:

Idea of Pointers:

- **Definition**: A pointer is a variable that stores the memory address of another variable.
- **Purpose**: Pointers are used for dynamic memory allocation, arrays, structures, and to facilitate the manipulation of data structures like linked lists and trees.
- **Advantages**:
  - Efficient memory management.
  - Allows for dynamic data structures.
  - Enables the creation of complex data structures such as linked lists, trees, and graphs.

Defining Pointers:

Syntax: The syntax for defining a pointer involves using the asterisk (*) symbol.

```
data_type *pointer_name;
```

Example:

```
int *ptr;   // ptr is a pointer to an integer
float *fptr; // fptr is a pointer to a float
char *cptr;  // cptr is a pointer to a character
```

Initialization: Pointers can be initialized to the address of a variable using the address-of operator (&).

```
        int var = 10;

        int *ptr = &var; // ptr now holds the address of var
```

Accessing Value Through Pointers

To access the value stored at the address a pointer points to, you use the dereference operator (*).

Example:

```
                int var = 20;

                int *ptr = &var;    // ptr points to var

                printf("Value of var: %d\n", *ptr); // Accessing value at ptr
```

Pointers can be incremented or decremented to navigate through memory addresses, especially when working with arrays.

Dereferencing: To access the value stored at the address a pointer is pointing to, the dereference operator (*) is used.

Example:

```
        value = *pointer;  // Access the value at the address stored in pointer
```

Pointer to a pointer:

A **pointer to a pointer** (or a **double pointer**) in C is a pointer that stores the address of another pointer. It allows for multiple levels of indirection, meaning you can refer to a value indirectly through more than one level of pointers.

Example:

```
int num = 10; // A normal integer variable

int *ptr = &num; // Pointer to an integer (stores the address of 'num')

 int **dbl_ptr = &ptr;
```

Pointer to Arrays:

Definition: A pointer to an array is a pointer that points to the first element of the array. It allows you to access and manipulate the array elements through pointer arithmetic.

Declaration: When declaring a pointer to an array, the pointer type must match the type of the array elements.

Example:

```
int arr[5] = {10, 20, 30, 40, 50};  // An array of integers

int *ptr = arr;  // Pointer to the first element of the array
```

Accessing Array Elements Using Pointers:

You can access array elements using pointers as follows:

- Using Pointer Arithmetic:

  ```
  printf("%d\n", *(ptr + 2));  // Outputs: 30 (3rd element)
  ```

  ```
  printf("%d\n", *(ptr + 2));  // Outputs: 30 (3rd element)
  ```

- Using Array Notation:

  ```
  printf("%d\n", ptr[3]);  // Outputs: 40 (4th element)
  ```

Both methods effectively access the same memory location, demonstrating the relationship between arrays and pointers.

Iterating Over an Array with Pointers:

You can use pointers to traverse an array efficiently:

Example:

```
for (int i = 0; i < 5; i++)
 {
printf("%d ", *(ptr + i));  // Outputs: 10 20 30 40 50
}
```

3k) Write a program for reading elements using a pointer into an array and display the values using the array.

Algorithm:

1. Start.
2. Declare an integer array and a pointer.
3. Initialize the pointer to point to the first element of the array.
4. Enter the number of elements in the array.
5. Use the pointer to read and store values in the array.
6. After storing the values, use the array directly to display the stored elements.
7. End.

// C program

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

```
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    int *ptr = arr;
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", (ptr + i));  // Using pointer arithmetic to store elements
    }
    printf("The elements in the array are:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);  // Using the array to display elements
    }
    return 0;
}
```

Pointer to Structures:

Definition: A pointer to a structure is a pointer that points to a structure variable, allowing you to access and modify its members using the pointer.

Structure Declaration:

```
struct Student {
    char name[50];
    int age;
    float grade;
};
```

Pointer Declaration: To declare a pointer to a structure

```
        struct Student *ptr;  // Pointer to a structure of type Student
```

Initializing Pointer to a Structure

You can initialize a pointer to a structure by assigning it the address of a structure variable

Example:

```
        struct Student s1 = {"Alice", 20, 85.5};
        struct Student *ptr = &s1;  // ptr points to s1
```

Accessing Structure Members Using Pointers

You can access members of a structure through a pointer using the arrow operator (->):

Example:

```
printf("Name: %s\n", ptr->name);  // Outputs: Alice
printf("Age: %d\n", ptr->age);    // Outputs: 20
printf("Grade: %.2f\n", ptr->grade);  // Outputs: 85.50
```

Modifying Structure Members Using Pointers:

You can also modify structure members using pointers:

Example:

```
ptr->age = 21;  // Changes age to 21
printf("Updated Age: %d\n", ptr->age);  // Outputs: 21
```

//C program of pointer to structure

```
#include <stdio.h>
#include <string.h>
// Define the structure
struct Student {
   char name[50];
   int age;
   float grade;
};

int main() {
   // Declare and initialize a structure variable
   struct Student s1;
   strcpy(s1.name, "Alice");  // Copy name into structure
   s1.age = 20;               // Set age
   s1.grade = 85.5;           // Set grade

   // Declare a pointer to the structure
   struct Student *ptr;
```

```
    // Initialize the pointer with the address of the structure variable
    ptr = &s1;

    // Access and print structure members using the pointer
    printf("Student Name: %s\n", ptr->name);
    printf("Student Age: %d\n", ptr->age);
    printf("Student Grade: %.2f\n", ptr->grade);

    // Modify structure members using the pointer
    ptr->age = 21;  // Update age
    ptr->grade = 90.0;  // Update grade

    // Print updated values
    printf("\nUpdated Student Information:\n");
    printf("Student Name: %s\n", ptr->name);
    printf("Student Age: %d\n", ptr->age);
    printf("Student Grade: %.2f\n", ptr->grade);
    return 0;
}
```

Self-Referential Structures:

Self-referential structures are structures that contain at least one member that is a pointer to the same structure type. This allows for the creation of complex data structures such as linked lists, trees, and graphs.

Example:

```
        struct Node
        {
          int data;             // Data part
        struct Node *next;     // Pointer to the next node
        };
```

In this example, the Node structure contains an integer data and a pointer next that points to another Node structure.

Characteristics of Self-Referential Structures:

- **Recursive Nature:** Self-referential structures allow for a recursive approach to data representation. Each instance of the structure can point to another instance of the same type.
- **Dynamic Data Structures:** These structures can grow and shrink in size, as they can be dynamically allocated and deallocated, making them suitable for scenarios where the number of elements is not known at compile time.

Usage of Self-Referential Structures in Linked Lists:

A linked list is a classic data structure that utilizes self-referential structures. It consists of nodes that are connected by pointers. Each node typically contains two components:

1. **Data Field:** Stores the actual data (e.g., an integer, string, or other types).
2. **Pointer Field:** Points to the next node in the list, allowing for the chaining of nodes.

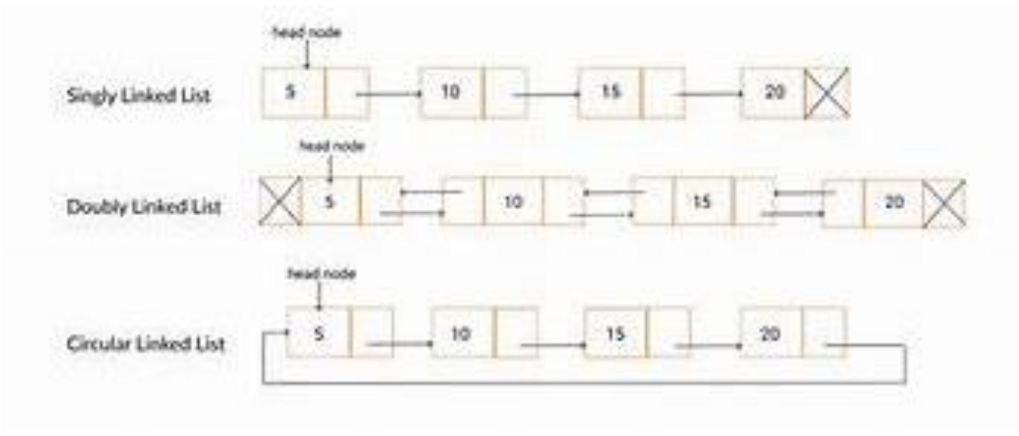
Basic Structure of a Linked List Node:

```
struct Node {
   int data;            // Data field
   struct Node *next;    // Pointer to the next node (address)
};
```

Types of Linked Lists

Linked lists can take several forms, each with its unique characteristics:

- **Singly Linked List:** Each node points to the next node in the sequence. The last node points to NULL, indicating the end of the list.
- **Doubly Linked List:** Each node contains two pointers—one pointing to the next node and another pointing to the previous node, allowing traversal in both directions.
- **Circular Linked List:** The last node points back to the first node, forming a circle. This can be applied to singly or doubly linked lists.



Operations on Linked Lists

Self-referential structures enable various operations on linked lists, including:

- **Insertion:** Adding a new node at the beginning, end, or any specified position in the list.
- **Deletion:** Removing a node from the list, which may require adjusting pointers to maintain the list structure.
- **Traversal:** Accessing each node in the list sequentially to perform operations like printing or searching for data.
- **Searching:** Finding a node with a specific value by traversing the list.

Advantages of Using Self-Referential Structures

- **Dynamic Size:** Unlike arrays, linked lists can grow or shrink dynamically as nodes are added or removed, which is memory efficient.
- **Ease of Insertion/Deletion:** Adding or removing nodes does not require shifting elements, as is necessary with arrays; only pointer adjustments are needed.
- **Flexibility:** They can easily represent complex data structures like trees and graphs, facilitating advanced algorithms and data manipulation.

3l) Write a program for display values reverse order from an array using a pointer.

Algorithm:

1. Start
2. Declare Variables:
    - An integer n to store the number of elements.
    - An integer array arr of size n.
    - An integer pointer ptr that points to the first element of the array.
3. Input the Number of Elements:

    - Prompt the user to enter the number of elements.
    - Read the value into n.
4. Input Array Elements:
    - Use a loop to input n elements into the array using scanf.
5. Display Array in Reverse Order:
    - Use a loop to iterate from n-1 to 0 and print the elements using pointer arithmetic (ptr + i).
6. End

// **C Program**

```
#include <stdio.h>
int main()
{
   int n;
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   int arr[n], *ptr;
   ptr = arr;
   printf("Enter %d elements:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", (ptr + i));
   }
   printf("The elements in reverse order are:\n");
   for (int i = n - 1; i >= 0; i--) {
      printf("%d\n", *(ptr + i));
   }
   return 0;
}
```

3m) Write a program through a pointer variable to sum of n elements from an array.

Algorithm:

1. Start
2. Declare Variables:
   - Declare an integer variable n to store the number of elements.
   - Declare an integer array arr of size n.
   - Declare an integer pointer ptr to point to the array.
   - Declare an integer variable sum to store the total sum of the elements.
3. Input the Number of Elements:
   - Prompt the user to enter the number of elements.
   - Read the value of n.
4. Input Elements:
   - Allocate memory for the array using arr[n].
   - Assign the address of the first element of arr to ptr.
   - Use a loop to read n elements into the array using pointer arithmetic.
5. Calculate the Sum:
   - Initialize sum to 0.
   - Use a loop to sum the elements using the pointer.
6. Display the Sum:
   - Print the total sum.
7. End.

//C Program

```
#include <stdio.h>
int main()
{
   int n,i,sum=0;
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   int arr[n],*ptr;
   ptr = arr;
   printf("Enter %d elements:\n", n);
```

```
    for (int i = 0; i < n; i++) {
        scanf("%d", ptr + i);
    }
    for (int i = 0; i < n; i++) {
        sum += *(ptr + i);
    }
    printf("The sum of the elements is: %d\n", sum);
    return 0;
}
```