1. **Differentiate between structure and union.**

Structure	**Union**
We use the struct statement to define a structure.	We use the union keyword to define a union.
Every member is assigned a unique memory location.	All the data members share a memory location.
Change in the value of one data member does not affect other data members in the structure.	Change in the value of one data member affects the value of other data members.
You can initialize multiple members at a time.	You can initialize only the first member at once.
A structure can store multiple values of the different members.	A union stores one value at a time for all of its members
A structure's total size is the sum of the size of every data member.	A union's total size is the size of the largest data member.
Users can access or retrieve any member at a time.	You can access or retrieve only one member at a time.

2.Write a program in C to find the average of N elements entered by a user using an array.

```
#include <stdio.h>

int main() {

    int n, i;

    float sum = 0.0, average;

    printf("Enter the numbers of data: ");

    scanf("%d", &n);

    float a[n];

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

        printf("Enter number %d: ", i+1);

        scanf("%f", &a[i]);

        sum += a[i];

    }

    average = sum / n;

    printf("Average = %.2f\n", average);

 return 0;}
```

Output:

```
Enter the numbers of data: 4
Enter number 1: 2
Enter number 2: 3
Enter number 3: 4
Enter number 4: 5
Average = 3.50
```

3. Write a program to store information of 10 students using structures. Information includes roll, name, marks of students.

```
#include <stdio.h>


struct Student {
    int roll;
    char name[50];
    float marks;
};


int main() {
    struct Student students[10];


    printf("Enter information of 10 students:\n");
    for (int i = 0; i < 10; i++) {
        printf("Enter roll, name, and marks for student %d: ", i+1);
        scanf("%d %s %f", &students[i].roll, students[i].name, &students[i].marks);
    }


    // Displaying information
    printf("\nInformation of 10 students:\n");
    for (int i = 0; i < 10; i++) {
        printf("Roll: %d, Name: %s, Marks: %.2f\n", students[i].roll, students[i].name,
students[i].marks);
```

```
    }

    return 0;
}
```

4. Explain nested structure with examples

- C provides us the feature of nesting one structure within another structure by using which, complex data types are created.
- For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code.
- Hence, to store the address of the employee, we need to store the address of the employee into a separate structure and nest the structure address into the structure employee. Consider the following program.

```
#include<stdio.h>
struct address
{
    char city[20];
    int pin;
    char phone[14];
};
struct employee
{
    char name[20];
    struct address add;
};
void main ()
{
    struct employee emp;
    printf("Enter employee information?\n");
    scanf("%s %s %d %s",&emp.name,&emp.add.city, &emp.add.pin, &emp.add.phone);
    printf("Printing the employee information....\n");
    printf("name: %s\nCity: %s\nPincode: %d\nPhone:
%s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
}
```

5. Create Structure Patient having ID, patient name and disease name as data member. WAP to read the details of 10 patients and print details of those patients having 'diabetes'.

```
#include<stdio.h>
#include<string.h>
struct patient{
char name[10],disease[10];
} pt[10] ;
int main(){
int i, n ;
char str[10]="diabetes";

printf("Enter the Number of Patients: ");
scanf("%d",&n);

for(i = 0 ; i < n ; i++) {
printf("\t\t\tPatient's Details:\n");

printf("\nEnter the First Name: ");
scanf("%s", pt[i].name);
printf("\nEnter the Disease Name: ");
scanf("%s", pt[i].disease);
}

printf(" \t Name \t Disease \n\n");
for(i = 0 ; i < n ; i++)

{

 int ans= strcmp(str, pt[i].disease);

 if(ans==0)

{

printf("\t %s \t  %s\n",  pt[i].name,pt[i].disease);

}

}

}
```

6. Write a program to check whether a square matrix is symmetric or not.

```
#include<stdio.h>

int main()
{

   int c, d, a[10][10], b[10][10], n, temp;
   printf("\nEnter the dimension of the matrix: \n\n");
   scanf("%d", &n);

   printf("\nEnter the %d elements of the matrix: \n\n",n*n);
   for(c = 0; c < n; c++) // to iterate the rows
      for(d = 0; d < n; d++) // to iterate the columns
         scanf("%d", &a[c][d]);

   // finding transpose of a matrix and storing it in b[][]
   for(c = 0; c < n; c++) // to iterate the rows
      for(d = 0; d < n; d++) //to iterate the columns
         b[d][c] = a[c][d];

   // printing the original matrix
   printf("\n\nThe original matrix is: \n\n");
   for(c = 0; c < n; c++)   // to iterate the rows
   {
      for(d = 0; d < n; d++)   // to iterate the columns
      {
         printf("%d\t", a[c][d]);
      }
   printf("\n");
   }

   // printing the transpose of the entered matrix
```

```
    printf("\n\nThe Transpose matrix is: \n\n");
    for(c = 0; c < n; c++) // to iterate the rows
    {
        for(d = 0; d < n; d++)   // to iterate the columns
        {
            printf("%d\t", b[c][d]);
        }
        printf("\n");
    }

    // checking if the original matrix is same as its transpose
    for(c = 0; c < n; c++)   // to iterate the rows
    {
        for(d = 0; d < n; d++)   // to iterate the columns
        {
            /*
                even if they differ by a single element,
                the matrix is not symmetric
            */
            if(a[c][d] != b[c][d])
            {
                printf("\n\nMatrix is not Symmetric\n\n");
                exit(0);    // a system defined method to terminate the program
            }
        }
    }

      printf("\n\nMatrix is Symmetric\n\n");
    return 0;
}
```

7. Write a program to accept a string from the user and check whether the string is palindrome or not without using the inbuilt string function.

```
#include <stdio.h>


int main() {

    char str[100];

    int i, len, flag = 1;


    printf("Enter a string: ");

    scanf("%s", str);
    // Calculate length of the string
    for (len = 0; str[len] != '\0'; len++);

    // Check for palindrome
    for (i = 0; i < len/2; i++) {
        if (str[i] != str[len-i-1]) {
            flag = 0;
            break;
        }
    }

    if (flag)
        printf("%s is a palindrome.\n", str);
    else
        printf("%s is not a palindrome.\n", str);

    return 0;
}
```

Output:

```
Enter a string: naman
naman is a palindrome.
```

8. Explain following function with proper example:-

strrev(), strcmp().gets(),strcpy(),strlen(), 2 library functions of math.h.

strrev():

- strrev() is a function used to reverse a given string.
- Syntax: char *strrev(char *str);
- Parameters:
    - str: This is the string to be reversed.
- Return Value: It returns a pointer to the reversed string.
- Example:

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

```
int main() {
    char str[] = "hello";
    printf("Original string: %s\n", str);
    printf("Reversed string: %s\n", strrev(str));
    return 0;
}
```

Output:

```
Original string: hello
Reversed string: olleh
```

strcmp():

- strcmp() is a function used to compare two strings.
- Syntax: int strcmp(const char *str1, const char *str2);
- Parameters:
    - str1: The first string to be compared.
    - str2: The second string to be compared.
- Return Value:
    - Returns 0 if both strings are equal.
    - Returns a negative value if str1 is less than str2.
    - Returns a positive value if str1 is greater than str2.
- Example:

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

int main() {
    char str1[] = "hello";
    char str2[] = "world";

    int result = strcmp(str1, str2);
    if (result == 0) {
        printf("Strings are equal\n");
    } else if (result < 0) {
        printf("str1 is less than str2\n");
    } else {
        printf("str1 is greater than str2\n");
    }
    return 0;
}
```

Output:

```
csharp
str1 is less than str2
```

gets():

- gets() is a function used to read a string from standard input (stdin).
- Syntax: char *gets(char *str);
- Parameters:
    - str: The buffer where the input string will be stored.
- Return Value: It returns the same buffer str.
- Note: gets() is considered unsafe because it doesn't perform any bounds checking, making it susceptible to buffer overflow vulnerabilities. It's recommended to use fgets() instead.

strcpy():

- strcpy() is a function used to copy one string to another.
- Syntax: char *strcpy(char *dest, const char *src);
- Parameters:
    - dest: The destination string where the content of src is to be copied.
    - src: The source string to be copied.
- Return Value: It returns a pointer to the destination string (dest).
- Example:

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

int main() {
    char src[] = "hello";
    char dest[10];

    strcpy(dest, src);
    printf("Copied string: %s\n", dest);
    return 0;
}
```

Output:

```
Copied string: hello
```

strlen():

- strlen() is a function used to determine the length of a string.
- Syntax: size_t strlen(const char *str);
- Parameters:
  - str: The string whose length is to be calculated.
- Return Value: It returns the length of the string (excluding the null-terminating character).
- Example:

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

int main() {
    char str[] = "hello";
    printf("Length of the string: %zu\n", strlen(str));
    return 0;
}
```

Output:

```
Length of the string: 5
```

These functions are part of the C Standard Library and are commonly used for string manipulation tasks.

As for the math.h library functions, it provides various mathematical functions like sin(), cos(), sqrt(), etc., for performing mathematical operations in C programming. Some of the common functions include:

1. sqrt(): Computes the square root of a number.
2. pow(): Computes the power of a number.
3. sin(), cos(), tan(): Computes the sine, cosine, and tangent of an angle, respectively.
4. fabs(): Computes the absolute value of a floating-point number.
5. ceil(), floor(): Rounds a floating-point number up or down to the nearest integer, respectively.
6. log(), exp(): Computes the natural logarithm and the exponential function, respectively.

These functions are useful for various mathematical calculations and are included in the math.h library in C.

9. **Explain compile time and runtime initialization of arrays with proper examples**

Compile Time Initialization:

- Compile time initialization involves specifying the array elements directly in the source code, typically at the point of declaration or using initialization lists.
- The compiler calculates the size of the array and allocates memory for it during the compilation phase.
- This method is efficient as it doesn't incur any overhead during program execution.
- Compile time initialization is suitable when the values of array elements are known at compile time.

Example in C:

```
#include <stdio.h>

int main() {

    // Compile time initialization

    int arr1[] = {1, 2, 3, 4, 5};

    // Accessing elements

    printf("arr1[0] = %d\n", arr1[0]); // Output: 1

    printf("arr1[1] = %d\n", arr1[1]); // Output: 2

    return 0;

}
```

Runtime Initialization:

- Runtime initialization involves populating the array with values during the execution of the program.
- The size of the array and the values of its elements are determined during runtime.
- This method is flexible as it allows dynamic allocation of memory and initialization based on user input or computation results.
- Runtime initialization is suitable when the size or values of array elements cannot be determined at compile time.

Example in C:

```
        #include <stdio.h>


        int main() {
            // Runtime initialization
            int size;
            printf("Enter the size of the array: ");
            scanf("%d", &size);
            int arr[size]; // Variable-length array (VLA)
            // Populate the array
            for (int i = 0; i < size; i++) {
                printf("Enter element %d: ", i+1);
                scanf("%d", &arr[i]);
            }
            // Accessing elements
            printf("Elements of the array:\n");
            for (int i = 0; i < size; i++) {
                printf("%d ", arr[i]);
            }
            return 0;
}
```

10. **Definition of array and static/dynamic initialization of 1D array.**
    An array in C is a collection of elements of the same data type stored in contiguous memory locations. Each element in the array is accessed using an index.

Static Initialization of a 1D Array in C:
In static initialization, the size and elements of the array are known at compile time.

```
#include <stdio.h>

int main() {
   // Static initialization of a 1D array
   int staticArray[5] = {1, 2, 3, 4, 5};

   // Accessing elements of the array
   for(int i = 0; i < 5; ++i) {
      printf("Element at index %d: %d\n", i, staticArray[i]);
   }

   return 0;
}
```

Dynamic Initialization of a 1D Array in C:
In dynamic initialization, the size of the array is determined at runtime.

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

int main() {
   int size;
   printf("Enter the size of the array: ");
   scanf("%d", &size);

   // Dynamic initialization of a 1D array
   int *dynamicArray = (int*)malloc(size * sizeof(int));

   // Check if memory allocation was successful
   if (dynamicArray == NULL) {
      printf("Memory allocation failed\n");
      return 1;
   }

   // Initializing elements of the array
   for(int i = 0; i < size; ++i) {
      dynamicArray[i] = i + 1;
   }

   // Accessing elements of the array
   for(int i = 0; i < size; ++i) {
      printf("Element at index %d: %d\n", i, dynamicArray[i]);
   }
```

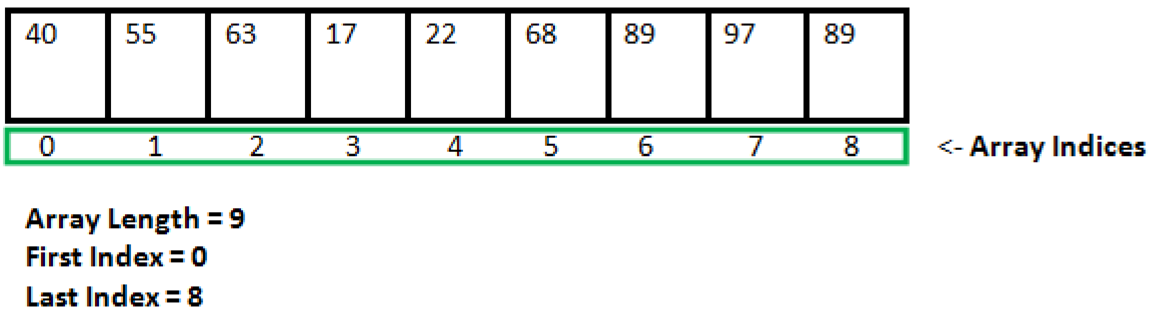
```
    // Free dynamically allocated memory
    free(dynamicArray);

    return 0;
}
```

In the dynamic initialization example, we use malloc to allocate memory for the array dynamically. We check if the allocation was successful, initialize the elements, and finally free the dynamically allocated memory using free to prevent memory leaks.

11. **Program to find the sum of even elements in an array:**

```
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int sum = 0;

    for (int i = 0; i < sizeof(arr) / sizeof(arr[0]); i++) {
        if (arr[i] % 2 == 0) {
            sum += arr[i];
        }
    }

    printf("Sum of even elements: %d\n", sum);

    return 0;
}
```

12. **Array definition, significance of array name, and array index:**
    - C language provides a capability that enables the user to design a set of similar data types, called array.
    - An array in C/C++ or in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array.
    - They can be used to store collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C/C++ can store derived data types such as the structures, pointers etc.
    -



```
#include <stdio.h>

int main() {
    // Declaration and initialization of an array
    int arr[5] = {10, 20, 30, 40, 50};

    // Accessing elements of the array
    printf("Element at index 0: %d\n", arr[0]); // prints 10
    printf("Element at index 2: %d\n", arr[2]); // prints 30

    // Modifying elements of the array
    arr[3] = 60; // changes the value at index 3 to 60

    // Accessing modified element
    printf("Element at index 3: %d\n", arr[3]); // prints 60

    return 0;
}
```

Arrays hold significant importance in the C programming language for various reasons:

Efficient Storage: Arrays in C provide a contiguous block of memory storage for elements of the same data type. This contiguous storage allows for efficient memory access and reduces memory overhead.

Random Access: Elements in an array can be accessed randomly using their index. This means that you can quickly access any element in the array without having to traverse through other elements. This property is essential for tasks like searching and sorting algorithms.

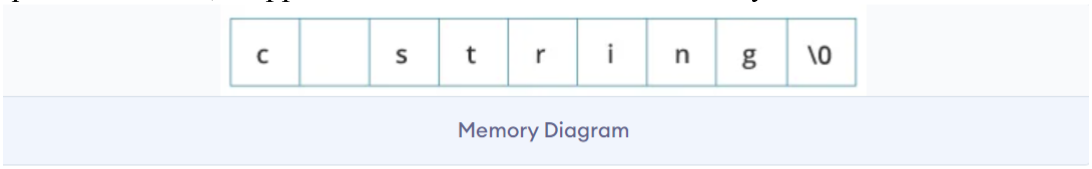
Iteration and Processing: Arrays are commonly used for iterating over a collection of elements. Loops, such as for loops, are frequently used to iterate over each element in an array for processing, manipulation, or computation.

Efficient Sorting and Searching: Arrays are fundamental for implementing efficient sorting and searching algorithms, such as bubble sort, quicksort, and binary search. These algorithms rely on random access to array elements for their performance.

Multi-dimensional Arrays: C supports multi-dimensional arrays, allowing you to represent data structures like matrices, tables, and grids. Multi-dimensional arrays are crucial for applications dealing with 2D or higher-dimensional data.

Dynamic Memory Allocation: In C, arrays can be dynamically allocated using functions like malloc() and calloc(). This allows for the creation of arrays whose size is determined at runtime, enabling more flexible memory management.

13. **What is a string? WAP that will read a word and rewrite it in alphabetical order.**
    - The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array. Character arrays are many a time also called strings.
    - A string constant is a one-dimensional array of characters terminated by a null ('\0').
        - char c[] = "c string";
    - When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.



Program that will read a word and rewrite it in alphabetical order.

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

int main() {
   char word[50];
   printf("Enter a word: ");
   scanf("%s", word);
   int len = strlen(word);

   // Sort the characters
   for (int i = 0; i < len - 1; i++) {
      for (int j = i + 1; j < len; j++) {
           if (word[i] > word[j]) {
               char temp = word[i];
               word[i] = word[j];
               word[j] = temp;
           }
      }
   }

   printf("Word in alphabetical order: %s\n", word);

   return 0;
}
```

14. Write a program to perform matrix multiplication by passing input matrix to the function and printing the resultant matrix.

```
#include <stdio.h>



// Function to multiply two matrices

void multiplyMatrices(int mat1[][10], int mat2[][10], int result[][10], int rows1, int
cols1, int rows2, int cols2) {

    // Check if matrices can be multiplied

    if (cols1 != rows2) {

        printf("Error: Number of columns in the first matrix must be equal to the
number of rows in the second matrix.\n");

        return;

    }



    // Perform matrix multiplication

    for (int i = 0; i < rows1; ++i) {

        for (int j = 0; j < cols2; ++j) {

            result[i][j] = 0;

            for (int k = 0; k < cols1; ++k) {
                result[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }
}

// Function to print a matrix
void printMatrix(int mat[][10], int rows, int cols) {
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            printf("%d\t", mat[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int mat1[10][10], mat2[10][10], result[10][10];
    int rows1, cols1, rows2, cols2;

    // Input dimensions of the first matrix
    printf("Enter the number of rows and columns of the first matrix: ");
    scanf("%d %d", &rows1, &cols1);

    // Input elements of the first matrix
```

```
printf("Enter elements of the first matrix:\n");
for (int i = 0; i < rows1; ++i) {
    for (int j = 0; j < cols1; ++j) {
        scanf("%d", &mat1[i][j]);
    }
}

// Input dimensions of the second matrix
printf("Enter the number of rows and columns of the second matrix: ");
scanf("%d %d", &rows2, &cols2);

// Input elements of the second matrix
printf("Enter elements of the second matrix:\n");
for (int i = 0; i < rows2; ++i) {
    for (int j = 0; j < cols2; ++j) {
        scanf("%d", &mat2[i][j]);
    }
}

// Perform matrix multiplication
multiplyMatrices(mat1, mat2, result, rows1, cols1, rows2, cols2);

// Print the resultant matrix
printf("Resultant matrix after multiplication:\n");
printMatrix(result, rows1, cols2);
```

```
            return 0;

        }
```

15. WAP to find the average of N elements entered by a user using an array.

```
#include <stdio.h>

int main() {

   int n, i;

   float sum = 0.0, average;

   printf("Enter the numbers of data: ");

   scanf("%d", &n);

   float a[n];

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

      printf("Enter number %d: ", i+1);

      scanf("%f", &a[i]);

      sum += a[i];

   }

   average = sum / n;

   printf("Average = %.2f\n", average);

 return 0;}
```

Output:

```
Enter the numbers of data: 4

Enter number 1: 2

Enter number 2: 3

Enter number 3: 4

Enter number 4: 5

Average = 3.50
```

16. WAP to find the transpose of a matrix using only one matrix.

```
#include <stdio.h>

int main() {

   int n, i;
```

```
    float num[100], sum = 0.0, avg;

    printf("Enter the numbers of elements: ");
    scanf("%d", &n);

    while (n > 100 || n < 1) {
      printf("Error! number should in range of (1 to 100).\n");
      printf("Enter the number again: ");
      scanf("%d", &n);
    }

    for (i = 0; i < n; ++i) {
      printf("%d. Enter number: ", i + 1);
      scanf("%f", &num[i]);
      sum += num[i];
    }

    avg = sum / n;
    printf("Average = %.2f", avg);
    return 0;
}
```

17. Explain pointers with examples. How an array is related to a pointer.

Pointers in C are variables that store memory addresses. They allow for dynamic memory allocation, efficient manipulation of data structures, and indirect access to memory locations. Pointers play a crucial role in C programming and are often used in conjunction with arrays, functions, and dynamic memory allocation.

Here's an explanation of pointers with examples:

Declaration and Initialization: Pointers are declared using the * operator. To declare a pointer to an integer, for example, you would write int *ptr;. Pointers can be initialized with the address of another variable using the address-of operator &. For example:

```
int x = 10;
int *ptr = &x; // Pointer ptr now holds the address of variable x
```

Dereferencing: Dereferencing a pointer means accessing the value stored at the memory address it points to. This is done using the dereference operator *. For example:

```
int x = 10;

int *ptr = &x;

printf("Value of x: %d\n", *ptr); //

Output: Value of x: 10
```

Pointer Arithmetic: Pointers can be manipulated using arithmetic operations such as addition and subtraction. When you perform arithmetic operations on pointers, they are adjusted by the size of the data type they point to. For example:

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

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



// Accessing elements of the array using pointer arithmetic

printf("First element: %d\n", *ptr); // Output: First element: 1

printf("Second element: %d\n", *(ptr + 1)); // Output: Second element: 2
```

Array and Pointer Relationship: In C, arrays are closely related to pointers. When you declare an array, its name acts as a pointer to the first element of the array. For example:

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

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

Both arr and ptr point to the same memory location, i.e., the address of the first element of the array. Therefore, you can use pointer arithmetic to access array elements.

```
printf("First element: %d\n", *ptr); // Output: First element: 1

printf("Second element: %d\n", *(ptr + 1)); // Output: Second element: 2
```

You can also use array indexing with pointers, which is equivalent to pointer arithmetic:

```
printf("Third element: %d\n", ptr[2]); // Output: Third element: 3
```

18. How do pointers differ from variables in C. Write a program to add 2 pointers.

Pointers and variables in C have distinct characteristics and serve different purposes:

Memory Address vs. Value:

Pointers store memory addresses, while variables store values.Pointers enable indirect access to memory locations, allowing manipulation of data at those locations.

Declaration and Initialization:

Pointers are declared with the * operator and initialized with the address of a variable.

Variables are declared with their respective data types and initialized with values.

Indirection:

Pointers can be dereferenced to access the value stored at the memory address they point to.

Variables directly hold values that can be accessed and manipulated without indirection.

Here's a program to add two pointers:

```
#include <stdio.h>

int main() {

   int x = 10, y = 20;

   int *ptr1 = &x; // Pointer to variable x

   int *ptr2 = &y; // Pointer to variable y

   int sum;

   // Add the values pointed to by ptr1 and ptr2

   sum = *ptr1 + *ptr2;

   // Print the result

   printf("Sum of values pointed to by ptr1 and ptr2: %d\n", sum);

   return 0;

}
```

19. Write a program to find a factorial of a given number using call by reference.

```
#include <stdio.h>

void findFactorial(int n, int *result) {

    *result = 1;

    for (int i = 1; i <= n; i++) {

        *result *= i;

    }

}

int main() {

    int num, factorial;

    printf("Enter a number: ");

    scanf("%d", &num);

    findFactorial(num, &factorial);

    printf("Factorial of %d is %d\n", num, factorial);

    return 0;

}
```

20. Explain the Dynamic memory allocation.

- The concept of dynamic memory allocation in c language enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.
- malloc()
- calloc()
- realloc()
- free()

malloc() function in C

- The malloc() function allocates single block of requested memory.
- It doesn't initialize memory at execution time, so it has garbage value initially.
- It returns NULL if memory is not sufficient.

- The syntax of malloc() function is given below:

  ```
  ptr=(cast-type*)malloc(byte-size)
  ```

  Example:

  ```
  int *ptr;

  ptr = (int*) malloc(100 * sizeof(int));
  ```

  Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte(base address) in the allocated memory.

```
#include<stdio.h>

#include<stdlib.h>

int main(){

int n,i,*ptr,sum=0;

printf("Enter number of elements: ");

scanf("%d",&n);

ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc

if(ptr==NULL)

{

printf("Sorry! unable to allocate memory");

exit(0);

}

printf("Enter elements of array: ");

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

{

scanf("%d",ptr+i);

sum+=*(ptr+i);
```

```
}

printf("Sum=%d",sum);

free(ptr);

return 0;

}
```

calloc() function in C

- The calloc() function allocates multiple block of requested memory.
- It initially initialize all bytes to zero.
- It returns NULL if memory is not sufficient.
- The syntax of calloc() function is given below:
- ptr=(cast-type*)calloc(number, byte-size)

realloc() function in C

- If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.
- the syntax of realloc() function.
- ptr=realloc(ptr, new-size)

free() function in C

- The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.
- Let's see the syntax of free() function.
- free(ptr)

21. Write a note on pointer initialization and dereferencing of pointer.

- **Pointer initialization** is the process where we assign some initial value to the pointer variable. We generally use the (&) addressof operator to get the memory address of a variable and then store it in the pointer variable.

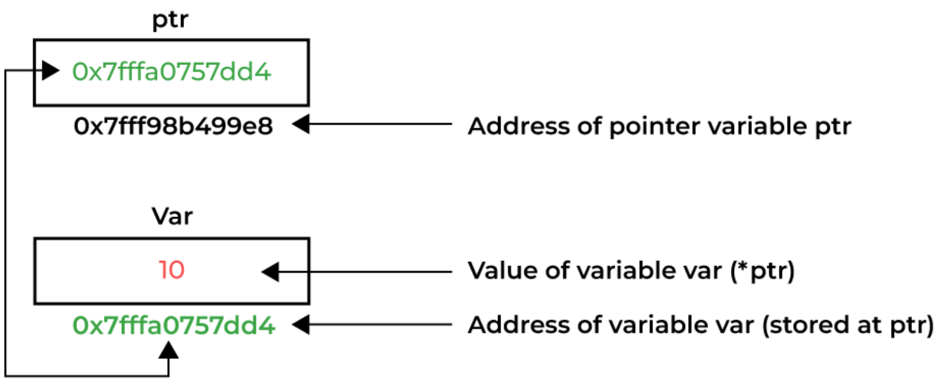
Example

```
int var = 10;
int * ptr;

ptr = &var;
```

We can also declare and initialize the pointer in a single step. This method is called pointer definition as the pointer is declared and initialized at the same time.

Example

```
int *ptr = &var;
```

- **Dereferencing a pointer** is the process of accessing the value stored in the memory address specified in the pointer. We use the same (*) dereferencing operator that we used in the pointer declaration.
- 
ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (stored at ptr)

22. Write a program to calculate the sum of series x-x/2!+x/3!-x/4!.....x/n

```
#include <stdio.h>

double factorial(int n) {

   if (n == 0 || n == 1) {

      return 1;

   }

   return n * factorial(n - 1);

}

int main() {

   double x, sum = 0.0;

   int n;
```

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

    scanf("%lf", &x);

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

    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {

        if (i % 2 == 0) {

            sum -= (x / factorial(i));

        } else {

            sum += (x / factorial(i));

        }

    }

    printf("Sum of the series: %.2lf\n", sum);

   return 0;

}
```

23. WAP to convert decimal numbers into any base.

```
#include <stdio.h>


// Function to convert decimal to any base

void decimalToBase(int num, int base) {

    // Array to store the digits of the result

    int result[100];

    int i = 0;


    // Convert decimal number to specified base

    while (num > 0) {

        result[i] = num % base;

        num = num / base;

        i++;
    }

    // Print the result in reverse order
    printf("Result in base %d: ", base);
    for (int j = i - 1; j >= 0; j--) {
        printf("%d", result[j]);
    }
    printf("\n");
}

int main() {
    int num, base;

    // Input decimal number and base
    printf("Enter a decimal number: ");
    scanf("%d", &num);
    printf("Enter the base to convert (2-16): ");
    scanf("%d", &base);

    // Check if base is valid
    if (base < 2 || base > 16) {
        printf("Invalid base. Please enter a base between 2 and 16.\n");
        return 1;
    }

    // Convert decimal to specified base
    decimalToBase(num, base);

    return 0;
}
```