Strings:

- The string can be defined as the one-dimensional array of characters terminated by a null ('\0').
- The character array or the string is used to manipulate text such as words or sentences. Each character in the array occupies one byte of memory, and the last character must always be 0.
- The termination character ('\0') is important in a string since it is the only way to identify where the string ends.
- When we define a string as char s[10], the character s[10] is implicitly initialized with the null in the memory.

There are two ways to declare a string in c language.

1. By char array
2. By string literal

1. BY char array

i) `char str[ ]={'c','o','d','i','n','g','A','g','e','\0'};`

i i) `char str[10 ]={'c','o','d','i','n','g','A','g','e'};`

```
int  s = sizeof(str) / sizeof(char);
```

```
for(int i=0;i<s;i++){
        printf("%c",str[i]);
 }
for(int i=0 ; str[i]!='\0' ; i++){
```

```
        printf("%c",str[i]);
  }
```

- How to declare string.

```
char   stringName [size] ;

int size=5;
char   str[size] ;
```

- How to take Input

1.

```
for(int i=0;i<s;i++){
        scanf(" %c",&str[i]);
 }
```

2.

```
 scanf("%s",str);
// can't take multiwords string input.
```

3. `gets( str );` // X

`gets( )` (Deprecated)

- Reads a string (line of text) from the standard input.
- **Warning**: `gets()` is unsafe because it does not check buffer boundaries and can lead to buffer overflows. Use `fgets()` instead.

4. **fgets(stringName, stringSize ,stdin);**

`fgets()`

- A safer alternative to `gets()`.
- Reads a line of text from the input up to a specified number of characters

5. **getchar( );**
//C getchar is a standard library function that takes a single input character from standard input.

- **How to take Output**

```
1. for(int i=0 ; str[i] != '\0'; i++){
        printf("%c",str[i]);
   }
```

```
2. printf("%s",str);
```

3. puts(str);

4. putchar(variable name);

//putchar that prints only one character to the standard output stream.

2. BY string literal

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

char   str[10] = "Coding Age";
```

- **String Function**

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

1. **`strlen()`** - Find the length of a string

Example:

```
int strlen( stringName );

char str[] = "Hello";
int size = strlen(str);
printf("Length: %d",size); //output : 5.
```

2. **`strcpy()`** - Copy one string into another

Example:

```
char* strcpy( stringName1 , stringName2 );

// Copy string from str2 to str1.

char src[] = "Source";
char dest[10];
strcpy(dest, src);
```

3. **`strcat()`** - Concatenate two strings

Example:

```
char* strcat( StringName1(dest) , stringName2(src) );
// Concatenate from  str2(source) to str1(destination )
```

```
char str1[20] = "Hello ";
char str2[] = "World!";

strcat(str1, str2);

// str1 = "Hello World!"
```

4. **`strcmp()` - Compare two strings**

Example:

```
int strcmp( StringName1 , stringName2 );
// Compare  Ascii value
 int value = strcmp(str1, str2);
```

Pointer:

- A pointer is a special type of variable in C that stores the memory address of another variable.
- Every variable in C is stored at a specific location in the computer's memory, and this location has a unique address.
- Pointers are powerful tools in C, enabling dynamic memory allocation, efficient array manipulation, and complex data structures.
- Pointers are declared using the * operator.

Syntax :

```
datatype  * var_Name ;

 int  *ptr ;

printf(" size of ptr = %d ",  sizeof(ptr));
                // pointer variable size = 8 bytes

 int  n=5;
 ptr=&n;

 printf("%p = %p",ptr,&n);

            //  0x7ffd8cdcdc64 = 0x7ffd8cdcdc64
              (memory address are hexadecimal value)
```

```
*ptr = n            ( both are same)
                    //  *ptr =5  & n=5
 ptr = &n;          (both are same)
                    //  ptr = 1000  and    &n = 1000
```

n address=1000

p

- ordinary variable size depends on data-types.
- pointer variable size not depends on data-types.
- pointer variable size| **2 byte** (16 bit)| **4 byte** (32 bit)| **8 bytes** (64 bit).

- & = address of operator . | Referencing operators. | unary operators.
- * = Indirection operator . | Dereferencing operators. | unary operators.

Referencing and Dereferencing Pointers:

Referencing a Pointer

- Referencing a pointer means obtaining the memory address of a variable. This is done using the address-of operator (&). When you reference a variable, you get a pointer that holds the address of that variable.

Dereferencing a Pointer

- Dereferencing a pointer means accessing the value stored at the memory address that the pointer holds. This is done using the dereference operator (*). When you dereference a
- pointer, you access or modify the value at the memory location pointed to by the pointer.
- **Referencing**: Using & to get the address of a variable.
- **Dereferencing**: Using * to access or modify the value at the address stored by a pointer.

Q. **How to Swap Two Numbers Using Pointers ?**

1. Call by Value:

- When you pass a variable to a function, a copy of the value is made.
- Any changes made inside the function won't affect the original value.

2. Call by Reference:

- Instead of passing a copy, you pass the memory address of the variable.
- Any changes made inside the function will affect the original value.

Pointers and Array .

```
    int  arr[5];
    int *ptr;
    ptr = arr;
    ptr = &arr[0]
/*
*ptr = arr[0];   |  *ptr+1 = arr[1];    |   *ptr+2 = arr[2];
|  *ptr+3 = arr[3];   |   *ptr+4 = arr[4]; |
 */
```

```
int *ptr;
int a[5];
for(int i=0;i<5;i++){
    a[i]=i+5;
}
ptr = a ;
for(int i=0;i<5;i++){
    printf("%d  ",*(ptr+i));
}
 printf( " %p=%p ",ptr,a);
```

Pointers and String.

```
    char str[20] = " Cager ";
    char *ptr;
        ptr = str;
        ptr = &str[0];

/*
 *ptr = str[0];   |  *ptr +1 = str[1];    | *ptr +2 = str[2];   |
*ptr +3 = str[3];   |   *ptr+4 = str[4]; |
 */
```

-

```c
#include <stdio.h>
int length(char *p) {
    int i=0;
    while(p[i]){
        i++;
    }
 return i;
}
int main() {
    char str[20]="Cagers";
    int l;
    l=length(str);
    printf("l=%d",l);
    return 0;
}
```

Double Pointer:

- A double pointer (also known as a pointer to a pointer) is a pointer that holds the address of another pointer.

- **Double pointer is declare**

```c
int **ptr;
```

Example :

```c
#include <stdio.h>
int main()
{
        int x = 10;
        int *p = &x ;                  // Pointer to x
        int **pp = &p;                 // Pointer to pointer p
```

```
    // Accessing values through double pointer
    printf("Value of x : %d\n", x );
    // Direct access
    printf("Value using *p: %d\n", *p);
    // Indirect access through pointer
    printf("Value using **pp: %d\n", **pp);
    // Indirect access through double pointer

  return 0;
}
```

Structure :

- The structure is a user-defined data type that allows you to combine data of different types under a single name.
- Each element inside a structure is called a **member**, and these members can have different data types.

- Structures are commonly used to represent complex data types like a record in a database or an object in real life.
- The keyword **struct** is used to define the structure.

Syntax of struct:

```
struct   StructureName        // struct is keyword
{
    dataType   member1;
    dataType   member2;
    // More members as needed
};
```

- This declaration is called a structure template or structure prototype.
- **No Memory Allocation:** The structure declaration itself does not allocate memory. It simply creates a template that tells the compiler what members exist in the structure and their respective types.
- **Memory Allocation Happens During Variable Declaration:** When you declare a variable of the structure type memory is allocated to hold the data for the members of that particular instance of the structure.

1. Structure Variable Declaration with Structure Template

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

struct  Person {
    char  name[30];    // cannot initialize members here.. Error
```

```
    int  age;
    float  height;
} person1 , person2  ;
```

Accessing Structure Members:

1. Using the Dot Operator

- When you have a structure variable, you access its members using the dot operator (.).

2. Using Pointers to Structures

- When using a pointer to a structure, you access its members using the arrow operator (->).

```
int main() {

    // person1 = 38 bytes | person2 = 38 bytes.
     person1 .age = 30;
     person1 .height = 6.1;
     strcpy(person1 .name, "Alice");

     // Access and Print Structure Members
     printf("Name: %s\n",person1 .name);
     printf("Age: %d\n",person1 .age);
     printf("height: %f\n", person1 .height);

return 0;
}
```

2. Structure Variable Declaration after Structure Template

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

struct  Person {    // Step 1: Define the structure template
```

```
    char name[30];
    int age;
    float height;
};
int main() {

    // Step 2: Declare structure variables
    struct Person person1;    // Declare a variable
    struct Person person2;

   // Also Declare multiple variables of type 'struct Person'
   //   struct Person person3 , person4;

     // Assign values to person1
     person1 .age = 26;       // scanf("%d",&person1.age);
     person1 .height = 6;    // scanf("%f",&person1.height);
     strcpy(person1 .name, "Abhi");

     printf("Person 1:\n");
     // Access and Print Structure Members
     printf("Name: %s\n",person1 .name);
     printf("Age: %d\n",person1 .age);
     printf("height: %f\n", person1 .height);

return 0;
}
```

Initialize Structure Members with zero:

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

struct Person {
    char name[30];
    int x;
```

```
    int y;
};
int main() {

    // Initialize the structure 'p' with all members set to zero
    struct Person p = {0};

    // Print the values to verify
    printf("Name: '%s'\n", p.name);    // Output: Name: ' '
    printf("x: %d\n", p.x);            // Output: x: 0
    printf("y: %d\n", p.y);            // Output: y: 0


return 0;
}
```

We can initialize structure members in 3 ways:

1. **Using the Assignment Operator**
2. **Using an Initializer List**
3. **Using a Designated Initializer List**

1. Using the Assignment Operator

- After declaring a structure variable, you can assign values to its members individually using the assignment operator (=).

2. Using an Initializer List

- You can initialize a structure at the time of declaration using an initializer list. This method sets the values of all members in the order they are declared.

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

struct  Person {
      char  name[30];
       int  age;
       float  height;
};
int main() {

         // Initialize structure with an initializer list
        struct Person person1= {"Ravi", 30, 5.7};

        // print the value
         printf("Name: %s\n",person1 .name);
         printf("Age: %d\n",person1 .age);
         printf("height: %f\n", person1 .height);

 return 0;
 }
```

3. Using a Designated Initializer List

- Designated initializers allow you to explicitly specify which member of the structure you are initializing.

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

```c
struct  Person {
      char  name[30];
       int  age;
       float  height;
};
int main() {

          // Initialize structure using designated initializers
          struct Person person1= {
                    .height = 5.7,
                    .age = 30,
                     .name = "Ravi",

          };


         // print the value
          printf("Name: %s\n",person1 .name);
          printf("Age: %d\n",person1 .age);
          printf("height: %f\n", person1 .height);

 return 0;
 }
```

typedef in C:

- `typedef` is a keyword used to create new type names (aliases) for existing types. It helps improve code readability and makes it easier to manage complex data types.

- **Basic Syntax:**

```
typedef  existingDataTypeName   newDataTypeName;
```

- Example:

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

typedef unsigned int uint;
        uint a=10000;
        printf("a=%d",a);

 return 0;
}
```

typedef for Structures:

- Basic Syntax:

```
typedef struct {
            // members
} NewTypeName;
```

- **Example:** Define a Struct with `typedef`

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

typedef struct {
        int id;
        char name[50];
} Student ;

int main() {

 Student  s1,s2;              // Declares a variable of type Student

       s1.id = 123;
       strcpy(s1.name, "Rahul");

       printf("ID: %d\n", s1.id);
       printf("Name: %s\n", s1.name);

 return 0;
 }
```

- **How to use Structure in Function:**

```c
#include <stdio.h>

 struct Employee {      // Define the structure
              int  id;
              char  name[50];
              float  salary;
};
```

```c
// Function to input data and return a structure
struct Employee inputEmployee() {

        struct Employee emp;
        printf("Enter employee ID: ");
        scanf("%d",  &emp.id);
        printf("Enter employee name: ");
        scanf(" %s",  emp.name);
        printf("Enter employee salary: ");
        scanf("%f", &emp.salary);

    return emp;

}

// Function to print structure data
void printEmployee( struct Employee emp) {

            printf("Employee Details:\n");
            printf("ID: %d\n",  emp.id);
            printf("Name: %s\n",  emp.name);
            printf("Salary: %.f\n",  emp.salary);
}

int main() {
        // Call the input function and get the structure
        struct Employee emp = inputEmployee();
            printEmployee(emp );    // Call the print function

 return 0;
}
```

- **Structure Array :**

Example :

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

struct Employee {     // Define the structure
        int  id;
        char  name[50];
        float  salary;
```

```
};

int main() {
    struct Employee emp[3]; // Step 2: Declare an Array of Structures

        // Step 3: Initialize and Use the Array
        // Initialize each element of the array
        emp[0].id = 1;
        strcpy(emp[0].name, "Alice");
        emp[0].salary = 50000;
        emp[1].id = 2;
        strcpy(emp[1].name, "Bob");
        emp[1].salary = 90.0;
        emp[2].id = 3;
        strcpy(emp[2].name, "Charlie");
        emp[2].salary = 78.5;
        // Print details of each student
        for (int i = 0; i < 3; i++) {
            printf("Employee Details:\n");
            printf("ID: %d\n",  emp[i].id);
            printf("Name: %s\n",  emp[i].name);
            printf("Salary: %.f\n",  emp[i].salary);
            printf("\n");
        }

 return 0;
}
```

- **Structure of Structure :**

Example :

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

// Define an Address structure
struct Address {
        char street[100];
```

```
        char city[50];
        int  zipCode;
};
// Define a Person structure that contains an Address structure
struct Person {
        char name[50];
        int age;
        struct Address address;
        // Nested structure
};

int main() {
    // Initialize a Person structure including the nested Address structure
    struct Person person1 = {
                    "John Doe", 30,
                    {"123 Main St", "New York", 10001},
    };
    // Accessing the outer structure members
    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);
    // Accessing the nested structure members
    printf("Street: %s\n", person1.address.street);
    printf("City: %s\n", person1.address.city);
    printf("Zip Code: %d\n", person1.address.zipCode);

 return 0 ;
}
```

Union :

- A **`union`** in C is a user-defined data type that allows different data types to be stored in the same memory location.
- The key difference between a `union` and a `struct` is that in a `struct`, each member has its own memory, while in a `union`, all members share the same memory space

- **Only One Member at a Time**: Since all members share the same memory location, only **one** member can hold a valid value at any given time. Writing to a new member will destroy the previous data.

Declaration and Syntax:

```
union UnionName {
        data_type1   member1;
        data_type2   member2;
};
```

- All members of the union share the same memory.
- The size of a union is determined by the size of its largest member.
- Only one member can be accessed at a time.

Example:

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

union Data {                    // Define a union
    int  i;
    float  f;
    char str[20];
};
int main() {
    // Declare a union variable
    union Data data;
    // Access and modify the union members one by one
    data.i = 10;
    printf("data.i: %d\n", data.i);
    data.f = 220.5;
```

```
    printf("data.f: %.1f\n", data.f);
    strcpy(data.str, "Hello");
    printf("data.str: %s\n", data.str);

 return 0;
}
```

Explanation:

1. When `data.i` is set to `10`, the memory location is used to store the integer value.
2. When `data.f` is assigned `220.5`, it overwrites the memory used by `data.i`.
3. When `strcpy(data.str, "Hello")` is executed, it again overwrites the memory space.
    - **Important:** The last written member is the one with a valid value, and all others may produce unexpected results if accessed afterward.

- **size of a union**

- The **size of a union** in C is determined by the size of its largest member, because all members share the same memory space. The total size of the union must be large enough to hold the largest member.
- **Example:**

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

union Example{              // Define a union

    int  i;                 // Typically 4 bytes
    double d;               // Typically 8 bytes (this is the largest)
```

```
    char c;                    // Typically 1 byte
};
int main() {
    // Declare a union variable
    union Data data;

    union Example data;
    printf("Size of union: %d bytes\n", sizeof(data));

    // Output: Size of union: 8 bytes
  return 0;
}
```

Difference between C Structure and C Union:

Structure	Union
Each member in a structure has its own memory space. The structure takes up enough space to hold all its members.	All members in a union share the same memory space. The union only takes up as much memory as its largest member.

You can use all the members at the same time.	You can only use one member at a time. Changing one member will overwrite the other.
The size is the total of all members.	The size is the largest member.
Use a structure when you need to store and use multiple values at the same time.	Use a union when you only need one value at a time, and you want to save memory.
It is declared using the struct keyword.	It is declared using the union keyword.

Pointer in Structure:

1. Pointer as a Member of a Structure

- A pointer can be used as a member of a structure to point to another variable or structure
- **Example:**

```
#include <stdio.h>

struct Person {
        char *name;
        int age;
};

int main() {
        char name[] = "John Doe";
        struct Person p;
// Assigning the address of the statically allocated array 'name' to the pointer
        p.name = name;
        p.age = 30;
        // Print values
        printf("Name: %s, Age: %d\n", p.name, p.age);
 return 0;
}
```

2. Pointer to a Structure

- You can also use pointers to structures.
- **Using Pointers to Structures :** When using a pointer to a structure, you access its members using the arrow operator (**->**).

Example:

```
#include <stdio.h>

struct Point {
        int x;
        int y;
};

int main() {
        struct Point p1 = {10, 20};

        struct Point *p;
        // Pointer to structure
        p= &p1 ;
        // Access members using pointer
        printf("Point: (%d, %d)\n", p->x, p->y);
      //printf("Point: (%d, %d)\n", (*p).x, (*p).y);
  return 0;
}
```

3.static linked lists or similar structures by manually linking nodes.

Example:(Linked List with Static Allocation):

```
#include <stdio.h>

struct Node {

        int data;
```

```
        struct Node *next;      // Pointer to the next node
};
int main() {
        // Statically allocate nodes
        struct Node n1, n2, n3;

        // Assign data and link nodes manually
        n1.data = 1;
        n1.next = &n2;
        n2.data = 2;
        n2.next = &n3;
        n3.data = 3;
        n3.next = NULL;

        struct Node *temp = &n1;
        // Print linked list
        while (temp != NULL) {
                printf("%d -> ", temp->data);
                temp = temp ->next;
        }
        printf("NULL\n");
return 0;
}
```

Dynamic Memory Allocation (DMA):

- The concept of dynamic memory allocation in c language enables the C programmer to allocate memory at runtime.
- in which the size of a data structure (like Array) is changed during the runtime.
- This provides flexibility to allocate memory as needed, based on program conditions, and deallocate it when it's no longer needed.
- It is particularly useful when dealing with data structures like linked lists, trees, etc.

- **Dynamic memory allocation in c language is possible by 4 functions of <stdlib.h> header file.**

1. **malloc()**
2. **calloc()**
3. **free()**
4. **realloc()**

- **The difference between static memory allocation and dynamic memory allocation.**

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
Stack Area memory allocation.	Heap Area memory allocation.
memory can't be increased while executing a program.	memory can be increased while executing a program.
used in an array.	used in the linked list.

`void*`

- `void*` is a pointer that can hold the address of any data type (e.g., `int`, `char`, `float`, etc.).
- A pointer that can point to any type of data and needs to be cast to the appropriate type before use.

* malloc() function in C:

Syntax of malloc():

```
void* malloc(size_t size);
```

- `int *arr = (int*) malloc(5 * sizeof(int));`

```
// Allocates memory for an array of 5 integers 20 bytes.
```

1.malloc() allocates a single large block of memory:

- The malloc() function is used to allocate a specified number of bytes in memory during program execution.
- The size of the block is specified by the argument passed to malloc(), which is the number of bytes you want to allocate.

2. Returns a void* pointer:

- malloc() returns a pointer of type void*, which means it can be cast to any pointer type (e.g., int*, char*, etc.).
- You typically cast this void* pointer to the type of data you plan to store in the allocated memory.

3. Does not initialize memory:

- One important feature of malloc() is that it does not initialize the memory block it allocates.
- This means the allocated memory contains whatever data was previously stored at that location, which are often referred to as garbage values.

4. Must free the memory:

- After the dynamically allocated memory is no longer needed, you should use the free() function to deallocate it and avoid memory leaks.

Example of malloc();

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int* ptr;    // This pointer will hold the base address of the block
created
    int n, i;

   printf("Enter number of elements:");
   scanf("%d",&n);                    // n= 5
   printf("Entered number of elements: %d\n", n);

   ptr = (int*)malloc(n * sizeof(int));  //Allocates memory of 20
bytes
        // Dynamically allocate memory using malloc()


// Check if the memory has been successfully allocated by malloc or not
  if (ptr == NULL) {
      printf("Memory not allocated.\n");
      return 0;
   }
   else {


 printf("Memory successfully allocated using malloc.\n");
```

```
        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    free(ptr);       // Frees the allocated memory

    return 0;
}
```

* calloc()(Contiguous Allocation) function in C:

- The calloc() function in C is used for dynamic memory allocation and is quite similar to malloc(). However, there are two key differences between calloc() and malloc():

1. **Memory Initialization to Zero:**

- calloc() initializes all allocated memory blocks to 0. This is different from malloc(),
- This is particularly useful when you want to ensure that the memory you allocate starts off in a clean, zeroed state.

2. **Takes Two Parameters:**
   - calloc() takes two arguments, whereas malloc() takes only one.
     - The first argument is the number of blocks (or elements) you want to allocate.
     - The second argument is the size of each block.

Syntax of calloc():

```
void *calloc(size_t num, size_t size);

int *arr = (int*) calloc(5 , sizeof(int));
// Allocates and initializes an array of 5 integers to zero
```

- **num: The number of elements to allocate.**
- **size: The size of each element in bytes.**

Example of calloc();

```c
#include <stdio.h>
#include <stdlib.h>
int main(){

   int* ptr;    // This pointer will hold the base address of the block created
   int n, i;

   printf("Enter number of elements:");
   scanf("%d",&n);                    // n= 5
   printf("Entered number of elements: %d\n", n);

   ptr = (int*)calloc(n , sizeof(int));
                    // Allocates and initializes an array of 5 integers to zero
         // Dynamically allocate memory using calloc()

// Check if the memory has been successfully allocated by calloc or not
 if (ptr == NULL) {
      printf("Memory not allocated.\n");
      return 0;
   }
   else {
printf("Memory successfully allocated using calloc.\n");

      // Get the elements of the array
      for (i = 0; i < n; ++i) {
         ptr[i] = i + 1;
      }

       printf("The elements of the array are: ");
      for (i = 0; i < n; ++i) {
         printf("%d, ", ptr[i]);
      }
   }
```

```
    free(ptr);      // Frees the allocated memory
    return 0;
}
```

* free() function in C:

- It is part of the standard library, declared in the <stdlib.h> header file.
- The free() method is used to **deallocate memory** that was previously allocated using functions like malloc(), calloc(), or realloc().
- whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax of free():

```
void free(void *ptr);

free( ptr );
```

* realloc() function in C:

- The realloc() function in C is used to resize a previously allocated block of memory. This function allows you to expand or shrink a dynamically allocated memory block without losing the data that was stored in it.

Resizing Memory:

- `realloc()` resizes the memory block pointed to by `ptr` to `new_size` bytes.
- If the new size is larger than the current size, the existing data is preserved, and additional memory is uninitialized (contains garbage values).
- If the new size is smaller, the extra memory is freed, and the remaining data is still preserved.

Memory Movement:

- If the new memory size can fit into the same memory block, `realloc()` does not move the block and simply changes its size.
- If the memory block cannot be resized in place, `realloc()` allocates a new block of memory, copies the old data to the new block, and frees the old memory block.

Null Pointer:

- If `ptr` is `NULL`, `realloc()` behaves like `malloc()` and allocates a new block of memory with the specified size.
- If `new_size` is 0 and `ptr` is not `NULL`, `realloc()` behaves like `free()` and frees the memory.

Return Value:

- `realloc()` returns a pointer to the newly allocated memory block. This pointer may be different from `ptr` if the block had to be moved to a new location.
- If the reallocation fails, it returns `NULL`, but the original memory block remains unchanged. Therefore, it is a good practice to

assign the result to a temporary pointer and check for NULL before modifying the original pointer.

Syntax of realloc():

```
void* realloc(void *ptr, size_t new_size);

arr = (int*) realloc(arr, 20 * sizeof(int));
// Resizes the memory block to hold 20 integers
```

Example of realloc();

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

   int *arr;
   int n = 5;
  // Initial memory allocation for 5 integers
    arr = (int*) malloc(n * sizeof(int));
     if (arr == NULL) {
    printf("Memory allocation failed!\n");
    return 0;
   }

    for (int i = 0; i < n; ++i) {
          arr[i] = i + 1;         // Initializing values
     }
 //size the array to hold 10 integers
   n = 10;
   arr = (int*) realloc(arr, n * sizeof(int));
```

```
    if (arr == NULL) {
        printf("Memory reallocation failed!\n");
        return 0;
    }
    // Initialize the new memory and print all values
    for (int i = 5; i < n; ++i) {
        arr[i] = i + 1;     // Initialize the new elements
    }
    // Print the array
    for (i = 0; i < n; ++i) {
        printf("%d, ", arr[i]);
    }

    free(arr);      // Free the allocated memory
    return 0;
}
```

2D Array :

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int rows = 3, cols = 4;
    // Step 1: Allocate memory for the array of row pointers
    int **arr =(int**) malloc(rows * sizeof(int *));
    if (arr == NULL) {
    printf("Memory allocation failed!\n");
    return 0;
    }
    // Step 2: Allocate memory for each row
    for (i = 0; i < rows; ++i) {
```

```
    arr[i]  = malloc(cols * sizeof(int));

    if (arr[i] == NULL) {
      printf("Memory allocation failed for row %d\n",i);
       return 0;
    }
  }
  int a=1;

  // Step 3: Initialize  the array

  for (i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; j++) {
       arr[i][j] = a;              // Example initialization
       a=a+1;
  } }

   // Step 4:  print the array

  for (i = 0; i < rows; ++i) {
      for (int j = 0; j < cols; j++) {
                printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
 // Step 5: Free the allocated memory

  for (i = 0; i < rows; ++i) {
    free(arr[i]);      // Free each row
  }

  free(arr);          // Free the array of row pointers
  return 0;
}
```

Explanation:

1. **Allocate row pointers**:

```
int **arr = malloc(rows * sizeof(int *));
```

This allocates memory for an array of `rows` pointers (each pointing to a row).

2. **Allocate each row**:
   Inside the loop, for each row, we allocate memory for `cols` integers using `malloc(cols * sizeof(int));`.
3. **Accessing Elements**:
   The elements can be accessed using `arr[i][j]` (like a regular 2D array).
4. **Freeing Memory**:
   It's important to free each row first, and then free the array of pointers.

File Handling:

File Handling in C

- File handling is an essential concept in C programming that allows programs to store data permanently in files and interact with them. Instead of entering data manually every time a program runs, file handling allows you to read data from or write data to a file automatically.
- Files are used to manage large amounts of data, save data permanently, and share data between different programs. In this explanation, we'll cover the importance of file handling, file modes, reading, writing, and closing files in detail.

---

1. What is a File?

A **file** is a collection of related data stored on a storage device, like a hard disk or SSD. It acts as a permanent storage medium for data. Files allow programs to save data that can be retrieved and used later.

Types of Files:

- **Text Files**: Store data as human-readable characters (e.g., `.txt`, `.csv`).
- **Binary Files**: Store data in raw binary form, which is compact and machine-readable (e.g., `.bin`, `.exe`).

What is File Handling?

File handling is the process of **creating, reading, writing, and closing files** in a program. It allows a program to interact with files stored on a computer's hard drive or any external storage.

Basic operations include:

- **Creating files**: Making a new file to store data.
- **Reading files**: Fetching data from a file.
- **Writing files**: Adding or updating content in a file.
- **Closing files**: Ensuring the program releases system resources after using the file.

Why is File Handling Useful?

1. **Persistent Storage**: Data stored in variables is lost when the program ends. File handling saves data permanently on disk, making it reusable.
2. **Large Data Management**: Files can store large amounts of data that may not fit in the system's RAM.
3. **Data Sharing**: Files allow different programs to share information (e.g., exporting data from one program for use in another).
4. **Debugging and Logging**: Programs can write logs or error messages to files, making it easier to troubleshoot issues.

---

3. Why Do We Learn File Handling?

Learning file handling is essential because:

1. **Data Permanence**: Variables store data temporarily files keep data even after the program ends.
2. **Data Sharing**: Files make it easy to share data between different programs or systems.
3. **Efficient Storage**: Large datasets can be stored and managed through files.
4. **Real-World Applications**: From text editors to games and databases, file handling is everywhere.

File Modes in C :

- In C, when working with files, you need to specify the mode in which you want to open the file. The file mode tells the system whether you want to read, write, or append to a file, and whether the file should be created or overwritten if it doesn't exist.

Syntax

```
FILE *fopen(const char *filename, const char *mode);
```

- `filename`: The name (and path, if necessary) of the file you want to open.
- `mode`: Specifies the purpose for opening the file (read, write, append, etc.).

1. "r" – Read Mode

- **Description**: Opens the file for **reading only**.
- **Behavior**:
    - If the file exists, it opens and lets you read from it.
    - If the file does **not exist**, the function returns NULL.
- **Use Case**: Reading data from an existing file.
- **Example**:

```
FILE *file = fopen("data.txt", "r");
```

2. "w" – Write Mode

- **Description**: Opens the file for **writing only**.
- **Behavior**:

- If the file exists, it **clears** its content (erases everything) and starts writing from the beginning.
- If the file does **not exist**, it **creates a new file**.

- **Use Case**: Writing new data, replacing any existing data.
- **Example**:

```
FILE *file = fopen("data.txt", "w");
```

3. "a" – Append Mode

- **Description**: Opens the file for **appending** new data at the end.
- **Behavior**:
  - If the file exists, the new data is written at the end of the file (the existing data is not erased).
  - If the file does **not exist**, it **creates a new file**.
- **Use Case**: Adding new data to the end of an existing file without losing previous data.
- **Example**:

```
FILE *file = fopen("data.txt", "a");
```

4. "r+" – Read and Write Mode

- **Description**: Opens the file for **both reading and writing**.
- **Behavior**:
  - If the file exists, you can read and write data.
  - If the file does **not exist**, the function returns NULL.
- **Use Case**: Modifying both the content and structure of an existing file (reading and writing).
- **Example**:

```
FILE *file = fopen("data.txt", "r+");
```

5. "w+" – Write and Read Mode

- **Description**: Opens the file for **both reading and writing**.
- **Behavior**:
  - If the file exists, it **clears** its content and starts writing from the beginning.
  - If the file does **not exist**, it **creates a new file**.
- **Use Case**: Reading and writing data when you want to start with an empty file.
- **Example**:

```
FILE *file = fopen("data.txt", "w+");
```

6. "a+" – Append and Read Mode

- **Description**: Opens the file for **both appending and reading**.
- **Behavior**:
  - If the file exists, you can read from it, and new data is appended at the end.
  - If the file does **not exist**, it **creates a new file**.
- **Use Case**: Reading from and appending data to the file without losing existing data.
- **Example**:

```
FILE *file = fopen("data.txt", "a+");
```

Summary of File Modes:

Mode	Operation	File Exists	File Doesn't Exist
"r"	Read-only	Opens the file	Returns NULL
"w"	Write (clears existing content)	Clears and writes	Creates a new file
"a"	Append (write to the end)	Appends to file	Creates a new file

"r+"	Read and write	Opens the file	Returns NULL
"w+"	Read and write (clears content)	Clears and writes	Creates a new file
"a+"	Read and append	Appends and reads	Creates a new file

Write to a File:

- When you need to write data to a file in C, there are several built-in functions that you can use. Each function serves a different purpose, whether it is writing individual characters, entire strings, or formatted data.

1. `fputc()` – Write a Single Character

- `fputc()` is used to write a single character to a file.

```
int fputc(char char, FILE *stream);
```

- `char`: The character to be written (represented as an integer).
- `stream`: The file pointer returned by `fopen()`

```c
#include <stdio.h>

int main() {
    FILE *file = fopen("data.txt", "w");     // Open file in read mode
    if (file == NULL) {
        printf("Error opening the file 'data.txt'\n");
        return 1;
    }

    fputc('A', file);          // Write the character 'A' to the file
    fputc('\n', file);         // Write a newline character

    fclose(file);                          // Close the file
    return 0;
}
```

- **fputc('A', file)** writes the character 'A' to the file.
- **fputc('\n', file)** adds a newline after writing 'A'.

2. fputs() – Write a String

- fputs() is used to write a string (an array of characters) to a file.

```c
int fputs(const char *str, FILE *stream);
```

- **`str`**: The string to be written.
- **`stream`**: The file pointer returned by `fopen()`.

```c
#include <stdio.h>

int main() {
    FILE *file = fopen("data.txt", "w");     // Open file in read mode
    if (file == NULL) {
        printf("Error opening the file 'data.txt'\n");
        return 1;
    }

    fputs("Hello, World!\n", file);      // Write a string to the file

     fclose(file);                       // Close the file
    return 0;
}
```

- `fputs()` does **not** automatically append a newline character, so you must include `\n` if you want one.

3. `fprintf()` – Write Formatted Data

- `fprintf()` is the most flexible writing function, allowing you to write formatted data (just like `printf()`, but to a file).

```c
int fprintf(FILE *stream, const char *format, var. name);
```

- **`stream`**: The file pointer returned by `fopen()`.
- **`format`**: A format string (similar to `printf()`).
- **`...`**: The values to be formatted and written.

```c
#include <stdio.h>

int main() {
    FILE *file = fopen("data.txt", "w");     // Open file in write mode
    if (file == NULL) {
        printf("Error opening the file 'data.txt'\n");
        return 1;
    }

    int number = 100;
    fprintf(file, "The number is: %d\n", number);
    // Write formatted data to the file

    fclose(file);                            // Close the file
    return 0;
}
```

- `fprintf()` is used when you need to write multiple types of data (e.g., integers, floats, strings) in a structured format.

Opening the File for Writing

Before using any writing function, you need to **open the file** using `fopen()` in the correct mode:

- `"w"` **(write mode)**: Overwrites the file if it exists, or creates a new file if it doesn't exist.
- `"a"` **(append mode)**: Adds data to the end of the file, preserving existing content.

```c
FILE *file = fopen("data.txt", "w");  // Open the file for writing
    if (file == NULL) {
                printf("Error opening the file 'data.txt'\n");
                return 1;
    }
```

Key Points for Writing Functions in C:

- `fputc()`: Writes one character at a time.
- `fputs()`: Writes an entire string (without adding a newline unless you specify it).
- `fprintf()`: Writes formatted data (like `printf()` but to a file).
- **Always close the file** with `fclose()` to ensure data is saved correctly.

Reading From a File:

- There are several ways to read from a file depending on the type of data you want to retrieve (character-by-character, line-by-line, or formatted).

1. Using `fgetc()` to Read Characters

- `fgetc()` reads **one character** from the file at a time.

```
char ch;
ch = fgetc(file);                    // Reads one character
printf("%c", ch);                    // Prints the character
```

- To read all characters in a file, you can loop until the **end of file** (EOF) is reached:

```
#include <stdio.h>

int main() {
    FILE *file = fopen("data.txt", "r");     // Open file in read mode
    if (file == NULL) {
        printf("Error opening the file 'data.txt'\n");
        return 1;
    }

    char ch;
```

```
    // Read and print each character from the file
   while ((ch = fgetc(file)) != EOF) {  // EOF is the end-of-file marker
            putchar(ch);                    // Print the character
    }

    fclose(file);                          // Close the file
   return 0;
}
```

- `fgetc(file)` reads one character at a time from `data.txt`.
- `putchar(ch)` prints the character to the screen.
- The loop continues until **EOF** (End of File) is reached.

2. Using `fgets()` to Read Lines

- `fgets()` reads a line of text from the file and stores it in a buffer.
  Here's an example that reads and prints each line from `data.txt`

```
#include <stdio.h>
int main() {
   FILE *file = fopen("data.txt", "r");     // Open file in read mode
   if (file == NULL) {
         printf("Error opening the file 'data.txt'\n");
         return 1;
   }


      char buffer[100];                    // Buffer to hold each line
      // Read and print each line from the file
      while (fgets(buffer, sizeof(buffer), file)) {
                  printf("%s", buffer);       // Print the line
   }

    fclose(file);                          // Close the file
   return 0;
}
```

- **fgets(buffer, sizeof(buffer), file)** reads up to 99 characters (or until a newline) from data.txt into buffer.
- **printf("%s", buffer)** prints the content of buffer.

3. Using `fscanf()` to Read Formatted Data

- fscanf() can read formatted data such as integers, strings, and floats. Here's an example that reads an integer and a string from data.txt:

```c
#include <stdio.h>

int main() {
    FILE *file = fopen("data.txt", "r");     // Open file in read mode
    if (file == NULL) {
        printf("Error opening the file 'data.txt'\n");
        return 1;
    }

    int number;
    char text[50];
    // Read an integer and a string from the file
    while (fscanf(file, "%d %s", &number, text) == 2) {
        printf("Number: %d, Text: %s\n", number, text);
    }

    fclose(file);                            // Close the file
    return 0;
}
```

- **fscanf(file, "%d %49s", &number, text)** reads an integer and a string from data.txt. %d reads an integer, and %49s reads a string up to 49 characters (leaving space for the null terminator).
- **printf("Number: %d, Text: %s\n", number, text)** prints the data read from the file.

Error Handling

- Always check if fopen() returns NULL to handle errors in file opening. Also, ensure to close the file using fclose() after you finish reading.

Summary of Reading Functions:

- **fgetc(FILE *fp)**: Reads a single character.
- **fgets(char *buffer, int size, FILE *fp)**: Reads a line or up to size-1 characters.
- **fscanf(FILE *fp, const char *format, ...)**: Reads formatted data.
- These functions provide flexible ways to read from a file, depending on your needs and the data format in data.txt.

Closing a File in C:

Why Closing a File is Important

1. **Saving Data**: If you write to a file, data is typically buffered (temporarily held in memory). The `fclose()` function flushes the buffer, ensuring all data is written to the disk.
2. **Releasing Resources**: Each open file consumes system resources. Closing the file frees up those resources.
3. **Avoiding Data Corruption**: If a program terminates or crashes before closing a file, any unsaved changes may be lost. Closing files ensures data integrity.
4. **Example:**

```
fclose(file);          // Close the file after writing
```

What is Binary Mode?

- **Binary mode** refers to a file access mode in which data is read and written as raw binary (0s and 1s) instead of human-readable

characters. In binary mode, data is stored exactly as it is in memory without any modifications or conversions.

Why Use Binary Mode?

1. **Efficiency**: Binary files are smaller because no extra conversions (like newline or EOF markers) occur.
2. **Accuracy**: Data remains intact without any alteration, making it ideal for storing complex structures.
3. **Compact Storage**: Ideal for storing images, executables, or raw data.

How to Use Binary Mode in C:

- Use **"rb"** to read a file in binary mode.
- Use **"wb"** to write a file in binary mode.
- Use **"ab"** to append data in binary mode.

---

1. Functions to Write Data to a File

In C, the following functions help **write data** to files:

- **`fwrite()` (used in binary mode):**
  - Writes raw binary data to a file.

```
struct Student {
    int id;
    char name[50];
    float marks;
} s = {1, "John Doe", 85.5};
```

```
FILE *fp = fopen("data.bin", "wb");

fwrite(&s, sizeof(s), 1, fp);

fclose(fp);
```

2. Functions to Read Data from a File

C provides the following functions to **read data** from files:

- `fread()` **(used in binary mode):**
  - Reads raw binary data from a file.

Example:

```
struct Student s;

FILE *fp = fopen("data.bin", "rb");

fread(&s, sizeof(s), 1, fp);

fclose(fp);
```

Binary Mode Example: Storing a Structure

Writing in Binary Mode:

```
#include <stdio.h>
struct Student {
    int id;
    char name[50];
    float marks;
};
void writeBinary() {
    struct Student s = {1, "Alice", 90.5};
```

```
    FILE *fp = fopen("student.bin", "wb");
    fwrite(&s, sizeof(s), 1, fp);  // Write binary
data
    fclose(fp);
}
```

Reading in Binary Mode:

```
void readBinary() {
    struct Student s;
    FILE *fp = fopen("student.bin", "rb");
    fread(&s, sizeof(s), 1, fp);  // Read binary
data
    fclose(fp);
    printf("ID: %d, Name: %s, Marks: %.2f\n",
s.id, s.name, s.marks);
}

int main() {
    writeBinary();
    readBinary();
    return 0;
}
```

Differences Between Text Mode and Binary Mode

Feature	Text Mode	Binary Mode
Data Format	Human-readable (e.g.,"123", "John")	Machine-readable (e.g., raw 01111000)
File Modes	"r", "w", "a"	"rb", "wb", "ab"
Efficiency	Larger size due to conversion	Smaller and faster
Use Case	Simple data (strings, numbers)	Complex data (structures, images, etc.)

File Seeking in C (fseek())

1. What is fseek()?

fseek() is a function in C that allows you to move the file pointer to a specific location within a file. It enables **random access** to file data instead of reading it sequentially from the beginning.

2. Why use fseek()?

- **Efficient Data Access**: Instead of reading the whole file, you can directly jump to the required position.
- **Modifying Specific Data**: You can update specific parts of a file without rewriting the entire content.
- **Skipping Unwanted Data**: Helps in skipping unnecessary data, reducing processing time.

3. How to use fseek()?

Syntax:

```
int fseek(FILE *file, long offset, int position);
```

- file: The file pointer.
- offset: Number of bytes to move (positive or negative).
- position: Reference point from where to move.

Values for position (Whence Argument)

Value	Description
SEEK_SET	Moves from the beginning of the file.
SEEK_CUR	Moves from the current position.
SEEK_END	Moves from the end of the file.

4. Examples of fseek() Usage

Example 1: Move to a Specific Position and Read Data

```
#include <stdio.h>

int main() {
    FILE *file = fopen("data.txt", "r");  // Open file in read mode
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fseek(file, 10, SEEK_SET);  // Move to the 10th byte from the start
    char ch = fgetc(file);      // Read the character at that position
    printf("Character at position 10: %c\n", ch);

    fclose(file);  // Close file
    return 0;
}
```

Example 2: Move to the End of the File and Get File Size

```
#include <stdio.h>

int main() {
    FILE *file = fopen("data.txt", "r");
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fseek(file, 0, SEEK_END); // Move to end of file
    long fileSize = ftell(file); // Get current position (file size)
    printf("File size: %ld bytes\n", fileSize);
```

```
    fclose(file);
    return 0;
}
```

Example 3: Using rewind() to Reset File Pointer

```
#include <stdio.h>

int main() {
    FILE *file = fopen("data.txt", "r");
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    fseek(file, 20, SEEK_SET);  // Move 20 bytes ahead
    rewind(file);  // Move back to the start of the file

    char ch = fgetc(file); // Read first character
    printf("First character after rewind: %c\n", ch);

    fclose(file);
    return 0;
}
```

5. Additional Functions for File Seeking

- ftell(FILE *file): Returns the current file pointer position.
- rewind(FILE *file): Moves the file pointer back to the beginning of the file.

6. Summary of File Seeking Functions

Function	Purpose

fseek(file, offset, position)	Moves the file pointer to a specific location.
ftell(file)	Gets the current position of the file pointer.
rewind(file)	Resets the file pointer to the beginning of the file.