

UNIT - V

Introduction to Pointers, dereferencing and address operators, pointer and address arithmetic, array manipulation using pointers, modifying parameters inside functions using pointers, Command line Arguments, Dynamic memory allocation, Null Pointer, generic pointer, dangling pointer

File Handling: Introduction to Files, Using Files in C, Reading from Text Files, Writing to Text Files, Random File Access.

Pointers

Introduction:

A pointer is a variable that stores the memory address of another variable.

Key Operators in Pointers:

1) Address-of Operator (&): Gives the memory address of a variable.

Example: `int *p = &x;`

2) Dereference Operator (*): Accesses the value at the address stored in the pointer.

Example: `int value = *p;`

Declaration and Initialization of Pointers:

1) Declaration:

`datatype *pointer_variablename;`

2) Initialization:

`pointer_variablename=&variable_name;`

Ex:

```
int x=101;
int *ptr;
ptr=&x;
```

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

```
int main()
```

```
{
```

```
    int x = 10;
```

```
    int *ptr = &x;                                // Pointer initialized with address of x
```

```
    printf("Address of x: %u\n", ptr);              // %u for pointer address
```

```
    printf("Value of x: %d\n", *ptr);              // Dereference to get value
```

```
    return 0;
```

```
}
```

Output:

Address of x: 6487600

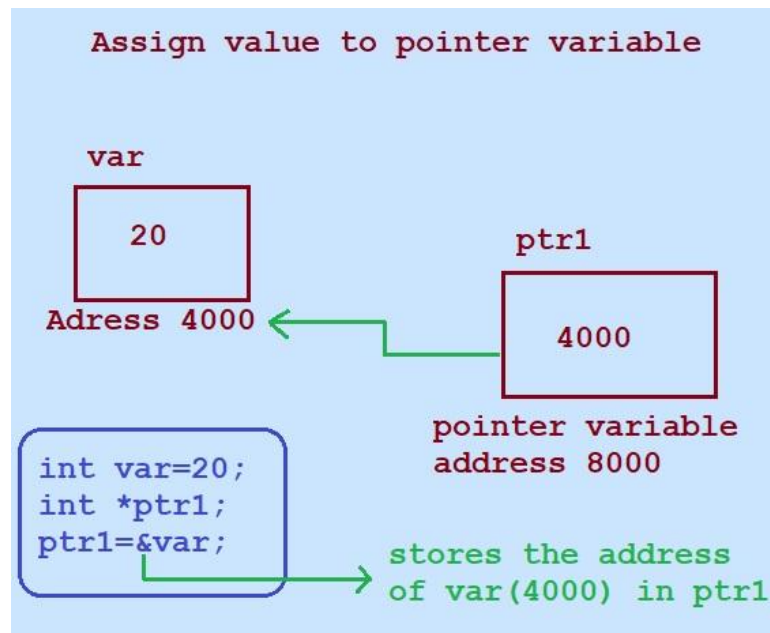
Value of x: 10

Explanation:

`int *ptr` declares a pointer to an integer,

`ptr = &x` stores the address of x in ptr.

`*ptr` accesses the value at that address.

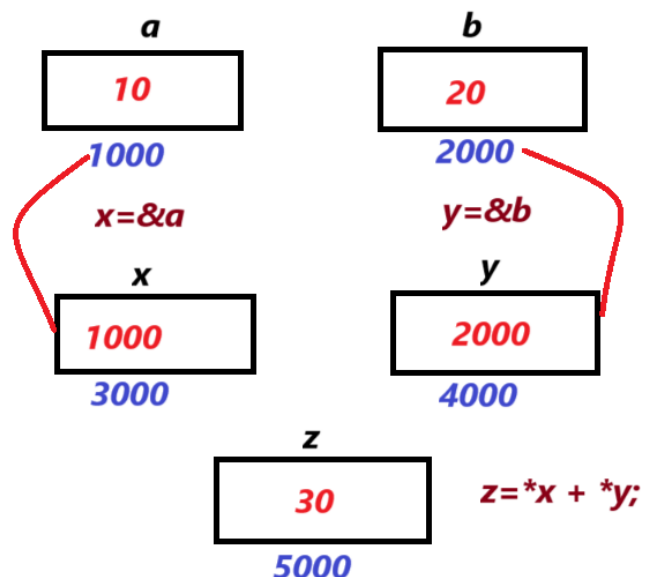


Program to find the addition of two numbers using pointers

```
#include<stdio.h>
int main()
{
    int a=10,b=20;
    int *x,*y;
    x=&a;
    y=&b;
    printf("a = %d b = %d\n",a,b);
    printf("%u %u\n",x,y);
    printf("%d %d\n",*x,*y);
    int z=*x + *y;
    printf("Sum = %d\n",z);
    return 0;
}
```

Output:

```
10 20
6487612 6487608
10 20
Sum = 30
```



Note: Storing the address of one data type in a pointer designed for another data type can lead to undefined behaviour or incorrect memory access.

i.e

```
int x=10;
char *ptr;
ptr=&x;           //Not recommended.
```

Advantages of pointers:

1) Direct Memory Access

Pointers allow direct access to memory locations, enabling efficient manipulation of memory and data.

2) Dynamic Memory Allocation

Pointers are essential for dynamic memory management using functions like `malloc()`, `calloc()`, `realloc()`, and `free()`.

3) Manipulation of Arrays and Strings

Pointers enable efficient traversal and manipulation of arrays and strings.

4) Building Complex Data Structures

Pointers are critical in constructing dynamic data structures like: Linked lists, Trees, Graphs, Queues and stacks.

5) Function Pointers

Pointers can store addresses of functions, enabling dynamic function calls and callback implementations.

6) Pointer to Pointer (Multilevel Pointers)

Enable creation of multi-dimensional arrays dynamically and handle complex data structures more effectively.

Accessing of Array elements with pointers:

In C, arrays and pointers are closely related. A pointer can be used to access and manipulate array elements because the array name itself acts as a pointer to the first element.

Accessing elements:

Using a pointer, array elements can be accessed by incrementing the pointer.

Pointer arithmetic:

Moving the pointer by 1 advance it to the next array element.

Example:

```
#include <stdio.h>
int main()
{
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr;           // Pointer to the first element of arr

    // Access array elements using the pointer
    for (int i = 0; i < 5; i++)
    {
        printf("Element %d: %d\n", i, *(ptr + i));
    }
    return 0;
}
```

Explanation:

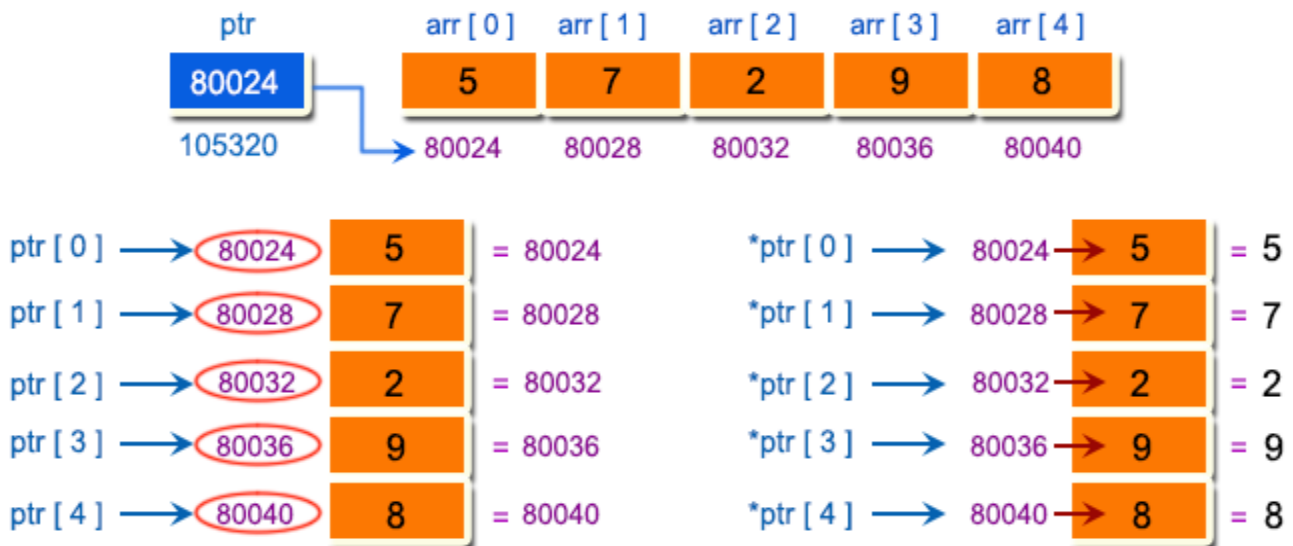
`ptr = arr`: `ptr` points to the first element (`arr[0]`).

`*(ptr + i)`: Accesses the `i`th element of the array by pointer arithmetic.

Pointer arithmetic ensures that `ptr + i` moves to the next integer in memory.

Output:

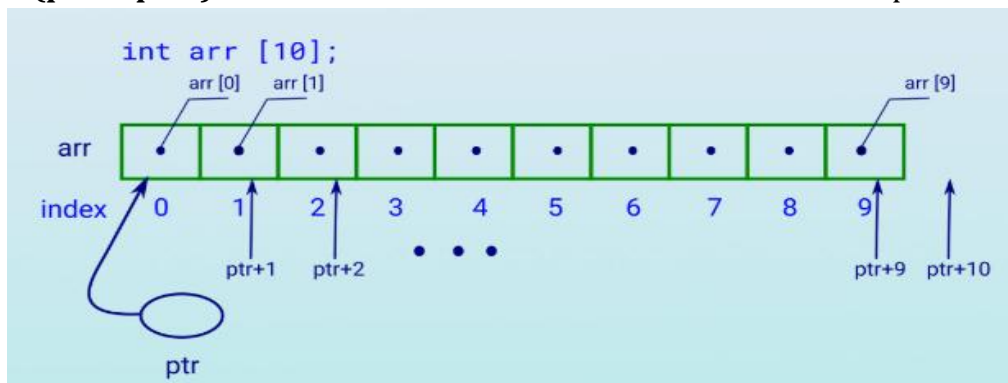
Element 0: 10
Element 1: 20
Element 2: 30
Element 3: 40
Element 4: 50



Pointer Arithmetic:

Pointer arithmetic refers to performing arithmetic operations like addition or subtraction on pointers. Since pointers store memory addresses, these operations move the pointer by the size of the data type it points to.

- 1) **Increment (ptr + 1):** Moves the pointer to the next element of the data type.
- 2) **Decrement (ptr - 1):** Moves the pointer to the previous element.
- 3) **Difference (ptr2 - ptr1):** Calculates the number of elements between two pointers



if ptr is points to the array then

$(ptr+i) \Rightarrow \text{Base Address of Array} + (i * \text{size_of_datatype})$

For Example

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

```
int *ptr;
```

```
ptr=arr;
```

// Pointer to the first element of arr

if ptr= 1000

[Assume Base address of arr is 1000]

$*(ptr+0) \Rightarrow 1000 + 0*4 \Rightarrow 1000$ i.e value at 1000 $\Rightarrow 10$
 $*(ptr+1) \Rightarrow 1000 + 1*4 \Rightarrow 1004$ i.e value at 1004 $\Rightarrow 20$
 $*(ptr+2) \Rightarrow 1000 + 2*4 \Rightarrow 1008$ i.e value at 1008 $\Rightarrow 30$

```
#include <stdio.h>
int main()
{
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr;           // Pointer to the first element of arr

    printf("Initial value: %d\n", *ptr); // Output: 10

    ptr++;                    // Move to the next integer (adds sizeof(int) to the address)
    printf("After increment: %d\n", *ptr); // Output: 20

    ptr += 2;                 // Move forward by 2 more elements
    printf("After adding 2: %d\n", *ptr); // Output: 40

    ptr--;                    // Move back by 1 element
    printf("After decrement: %d\n", *ptr); // Output: 30
    return 0;
}
```

Output:

Initial value: 10
After increment: 20
After adding 2: 40
After decrement: 30

Explanation:

ptr++ moves the pointer from arr[0] to arr[1].
ptr += 2 moves it from arr[1] to arr[3].
ptr-- moves it back to arr[2].

Program to print the addresses of variables and arrays declared in a program

```
#include<stdio.h>
int main()
{
    int x=10,i;
    printf("x = %d\n",x);
    printf("Address of x = %u\n",&x);
    char ch='A';
    printf("Address of ch = %u\n",&ch);

    printf("Addresses of integer array elements are\n");
    int arr[5]={11,22,33,44,55};
```

```

for(i=0;i<5;i++)
{
    printf("%u ",&arr[i]);
}
printf("Addresses of character array elements are\n");
char chrarr[3]={'H','i','\0'};
printf("\n");
for(i=0;i<2;i++)
{
    printf("%u ",&chrarr[i]);
}
return 0;
}

```

Output:

x = 10

Address of x = 6487624

Address of ch = 6487623

Addresses of integer array elements are

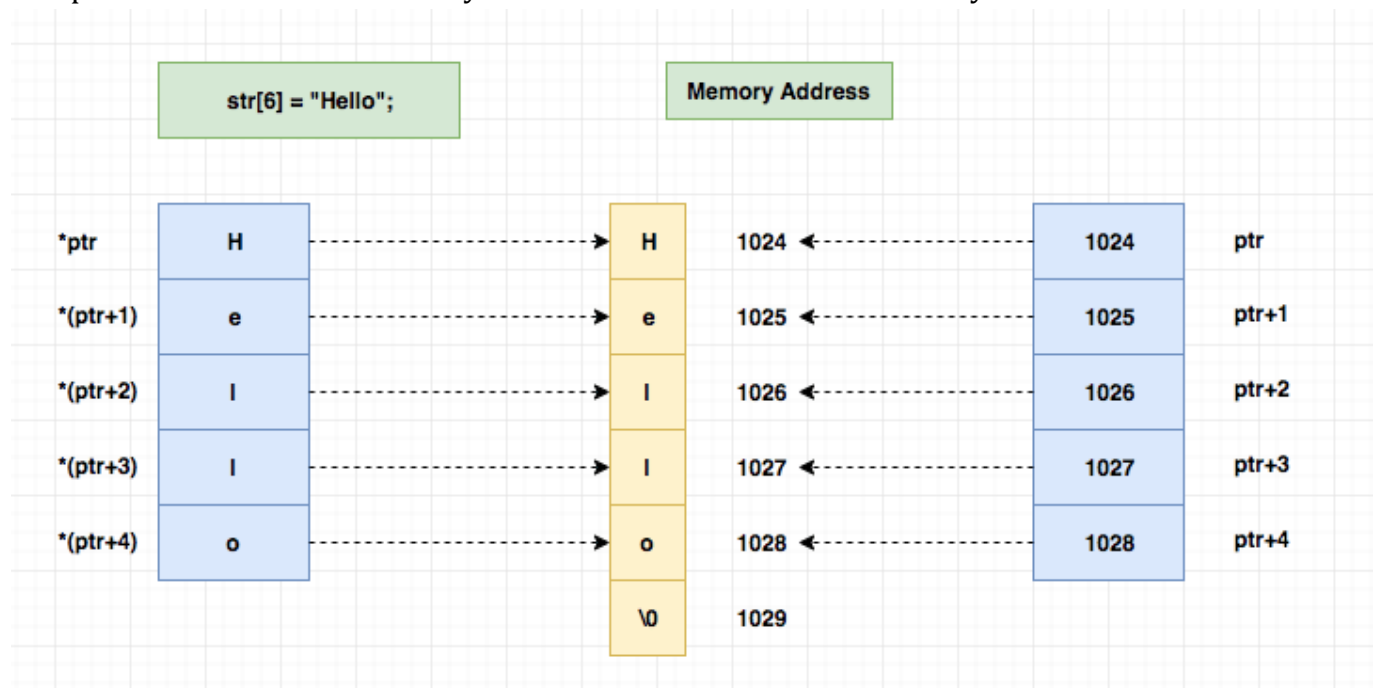
6487600 6487604 6487608 6487612 6487616

Addresses of character array elements are

6487584 6487585

Pointers with Character Array:

Pointers can be used to access elements of a character array, which is often used to store strings. The pointer can traverse the array to access each character individually.



```
#include <stdio.h>
int main()
{
    char str[] = "Hello, World!";
    char *ptr = str;                // Pointer to the first character of str

    // Access each character using the pointer
    while (*ptr != '\0')
    {
        // '\0' marks the end of the string
        printf("%c ", *ptr);        // Print the character
        ptr++;                      // Move to the next character
    }
    return 0;
}
```

Output:

Hello, World!

Explanation:

ptr = str: ptr points to the first character 'H'.

*ptr: Dereferences the pointer to get the current character.

ptr++: Moves the pointer to the next character in the array.

'\0': Indicates the end of the string.

Parameter passing techniques:

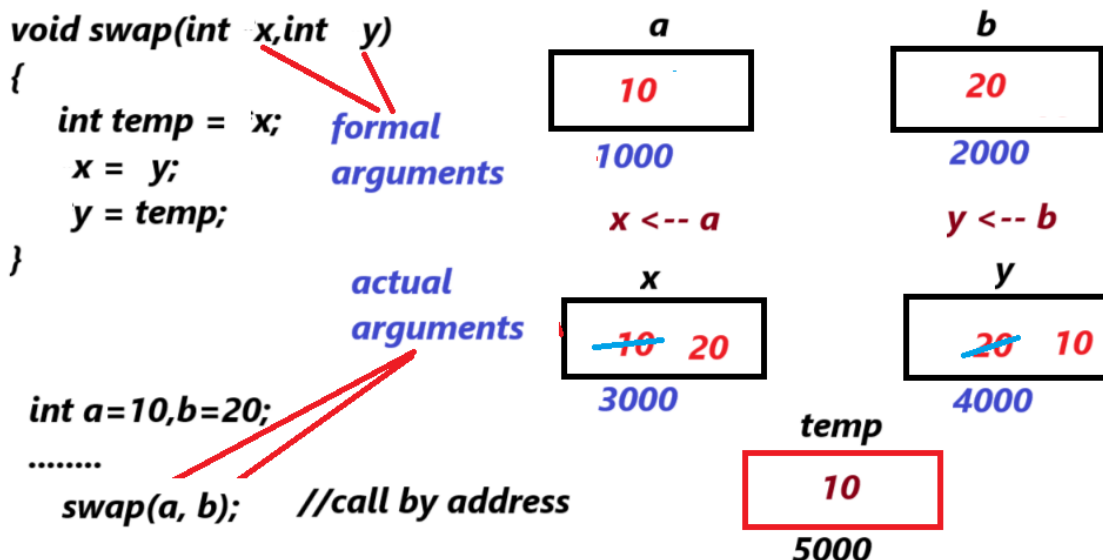
Parameter passing techniques determine how arguments are passed to functions in programming.

1. Pass by Value or Call by Value:

A copy of the actual argument is passed to the function. Changes made inside the function do not affect the original variable.

- There are two copies of parameters stored in different memory locations.
- One is the original copy and the other is the function copy.
- Any changes made inside functions are not reflected in the actual parameters of the caller.

Call by Value : Changes in formal parameters not reflected in actual arguments



Ex: Program to find the swapping of two numbers using call by value

```
#include<stdio.h>
void swap(int,int);
void swap(int x,int y)
{
    int temp=x;
    x=y;
    y=temp;
}
int main()
{
    int a,b;
    printf("Enter any two values\n");
    scanf("%d%d",&a,&b);
    printf("Before swapping a and b values\n");
    printf("%d %d\n",a,b);
    swap(a,b);           // Call by Value
    printf("After Swapping of a and b values\n");
    printf("%d %d\n",a,b);
    return 0;
}
```

Output:

```
Enter any two values
10 20
Before swapping a and b values
10 20
After Swapping of a and b values
10 20
```

Note:

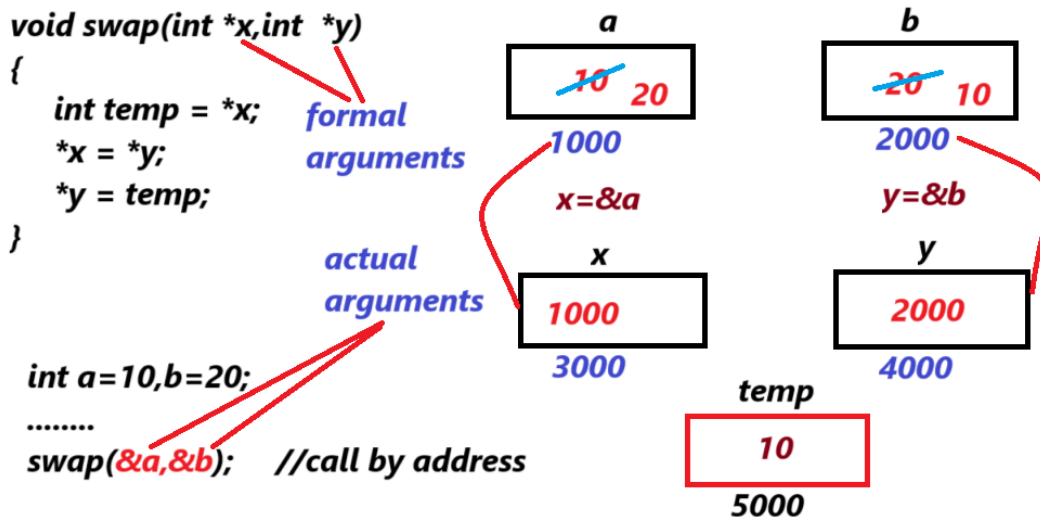
In Call by value whatever modification done on formal arguments will not shown effect on original arguments.

2. Pass by Address or Call by Address:

In pass by Address, the address of the variable is passed, allowing the function to modify the original variable.

- Both the actual and formal parameters refer to the same locations.
- Any changes made inside the function are actually reflected in the actual parameters of the caller.

Call by address: Changes in formal parameters reflected in actual arguments



Ex: Program to find the swapping of two numbers using call by address.

```
#include<stdio.h>
void swap(int *,int *);
void swap(int *x,int *y)
{
    int temp=*x;
    *x=*y;
    *y=temp;
}
int main()
{
    int a,b;
    printf("Enter any two values\n");
    scanf("%d%d",&a,&b);
    printf("Before swapping a and b values\n");
    printf("%d %d\n",a,b);
    swap(&a,&b);    //Call by Address
    printf("After Swapping of a and b values\n");
    printf("%d %d\n",a,b);
    return 0;
}
```

Output:

```
Enter any two values
10 20
Before swapping a and b values
10 20
After Swapping of a and b values
20 10
```

Note:

In Call by Address whatever modification done on formal arguments will show that effect on Actual arguments.

Command line Arguments:

Command-line arguments allow users to pass input values to a C program when it is executed. These arguments are passed to the main() function via two parameters: **argc** and **argv**.

Syntax of main Function with Command-Line Arguments:

```
int main(int argc, char *argv[])
```

where

argc (Argument Count):

- An integer representing the number of command-line arguments passed.
- Includes the name of the program, so argc is always at least 1.

argv (Argument Vector):

- An array of character pointers (strings) representing the actual arguments passed.
- argv[0] contains the program's name.
- argv[1] to argv[argc-1] contain the arguments provided by the user.

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

```
int main(int argc, char *argv[])
```

```
{
    printf("Number of arguments: %d\n", argc);

    for (int i = 0; i < argc; i++)
    {
        printf("Argument %d: %s\n", i, argv[i]);
    }
    return 0;
}
```

Output:

```
C:\Users\mcnu5\OneDrive\Documents>Commandline1.exe 3581 Srinu_M THUB
```

```
Number of arguments: 4
```

```
Argument 0: Commandline1.exe
```

```
Argument 1: 3581
```

```
Argument 2: Srinu_M
```

```
Argument 3: THUB
```

Program to find the Sum of Two Numbers Passed as Command-line Arguments

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

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

```
int main(int argc, char *argv[])
```

```
{
    if (argc != 3)
    {
        printf("Usage: %s num1 num2\n", argv[0]);
        return 1;
    }
    int num1 = atoi(argv[1]);
    int num2 = atoi(argv[2]);
    int sum = num1 + num2;
```

```
printf("Sum: %d\n", sum);
return 0;
}
```

Output:

```
C:\Users\mcnu5\OneDrive\Documents>Commandline2.exe 25 35
Sum: 60
```

Null Pointer:

A **null pointer** in C is a pointer that doesn't point to any valid memory location. It is often used to signify that the pointer is not currently referencing any data.

In C, the macro NULL represents a null pointer constant.

Syntax:

```
int *ptr = NULL;
```

Here, ptr is a pointer that does not point to any memory location.

Example:

```
#include <stdio.h>
int main() {
    int *ptr = NULL;                // Initialize a pointer to NULL
    if (ptr == NULL)
    {
        printf("Pointer is null.\n");
    }
    else
    {
        printf("Pointer is not null.\n");
    }
    return 0;
}
```

Output:

Pointer is null.

Dangling Pointer:

A **dangling pointer** is a pointer that references a memory location that has been **freed, deleted, or deallocated**, meaning the memory is no longer valid or accessible. Dereferencing a dangling pointer results in undefined behaviour, often leading to crashes or memory corruption.

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr = (int *)malloc(sizeof(int));    // Allocate memory
    *ptr = 42;                                // Assign a value
    printf("Value: %d\n", *ptr);

    free(ptr);                                // Free the allocated memory
}
```

```
// ptr is now a dangling pointer
printf("Accessing freed memory: %d\n", *ptr); // Undefined behaviour
return 0;
}
```

Generic Pointer:

A **generic pointer** in C is a pointer of type `void *` that can hold the address of any data type. Unlike other pointers (e.g., `int *`, `char *`), a generic pointer does not have a specific data type associated with it, making it versatile for handling various types of data.

Syntax:

```
void *ptr;
```

Example:

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

```
int main()
{
    int num = 10;
    void *ptr;                // Declare a generic pointer

    ptr = &num;               // Store the address of an integer

    // Cast to int * before dereferencing
    printf("Value: %d\n", *(int *)ptr);
    return 0;
}
```

Output:

Value: 10

Dynamic Memory Allocations in C:

Dynamic memory allocation in C allows the program to allocate memory at runtime (instead of compile-time) using functions from the standard library. This flexibility is crucial when the exact size of the required memory is not known in advance.

Why Use Dynamic Memory Allocation?

1. **Efficient Memory Use:** Allocate only the memory needed at runtime.
2. **Flexible Arrays:** Create variable-sized arrays.
3. **Data Structures:** Used for dynamic data structures like linked lists, trees, stacks, and queues.

Key Functions for Dynamic Memory Allocation

Function	Description	Returns
<code>malloc()</code>	Allocates memory block of specified size (in bytes)	<code>void *</code>
<code>calloc()</code>	Allocates and initializes memory block	<code>void *</code>
<code>realloc()</code>	Reallocates memory block to new size	<code>void *</code>
<code>free()</code>	Deallocates previously allocated memory	<code>void</code>

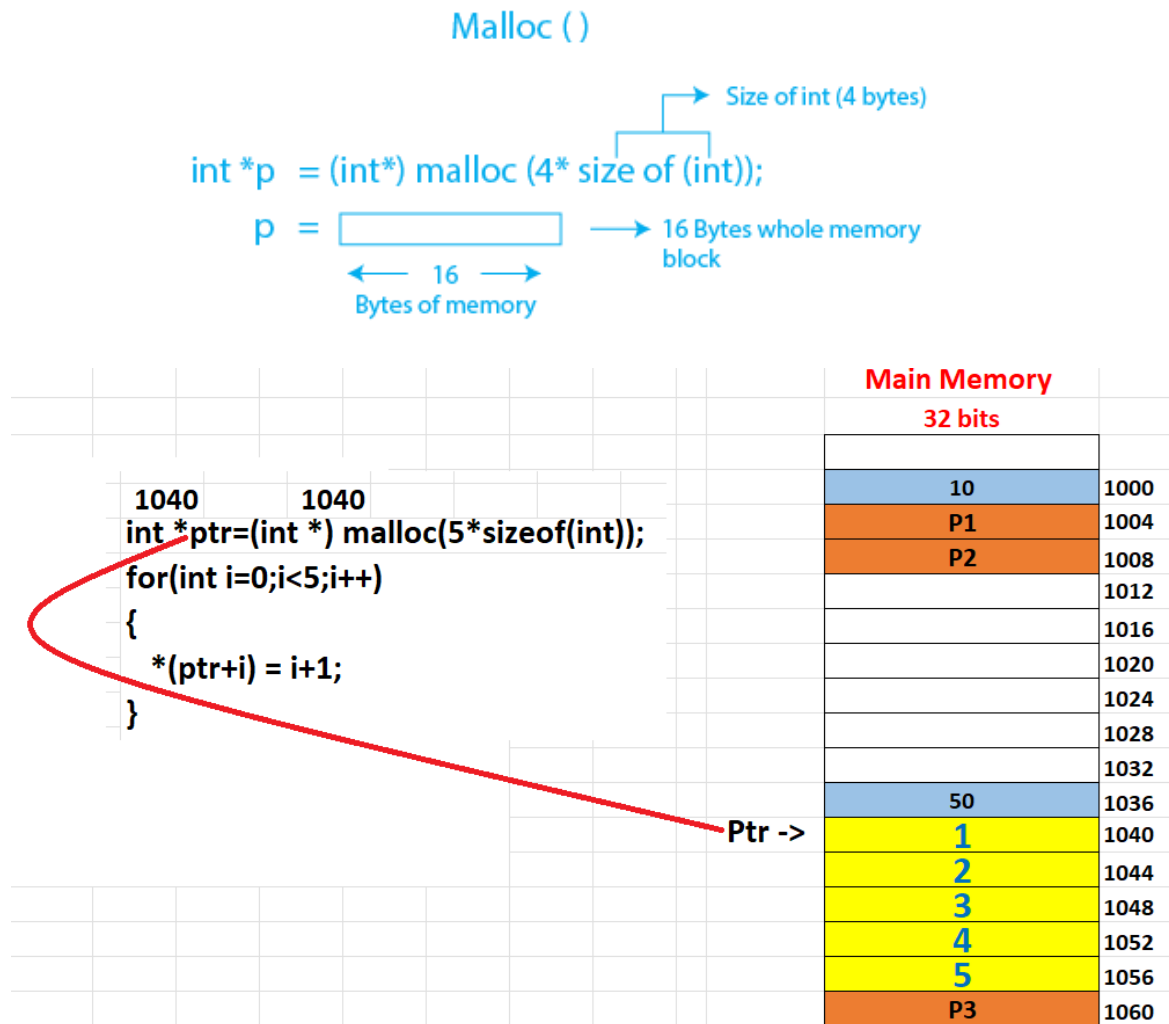
1) **malloc()**: malloc() is a standard library function in C that dynamically allocates a block of memory on the heap. The allocated memory remains uninitialized, meaning it contains garbage values.

Syntax:

void *malloc(size_t size);

size: Number of bytes to allocate.

Returns: A void * pointer to the allocated memory. If memory allocation fails, malloc() returns NULL.



Example:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n = 5;
    // Allocate memory for 5 integers
    int *ptr = (int *)malloc(n * sizeof(int));

    if (ptr == NULL)
    {
        printf("Memory allocation failed.\n");
        return 1;
    }
    // Exit the program if memory allocation fails
}
```

```

// Initialize and display the array elements
for (int i = 0; i < n; i++)
{
    ptr[i] = i + 1;                // Assign values
}
printf("Allocated and initialized array:\n");
for (int i = 0; i < n; i++)
{
    printf("%d ", ptr[i]);
}
printf("\n");

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

```

Output:

Allocated and initialized array:

1 2 3 4 5

- 2) **calloc()**: calloc() is a standard library function in C used to dynamically allocate memory for an array. It initializes all allocated memory to zero, unlike malloc(), which leaves the memory uninitialized.

Syntax:

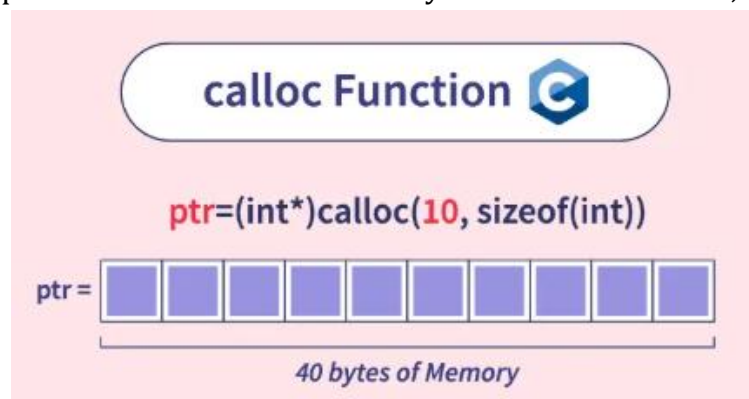
```
void *calloc(size_t num, size_t size);
```

Where

num: Number of elements to allocate.

size: Size of each element in bytes.

Returns: A void * pointer to the allocated memory. If the allocation fails, it returns NULL.



Example:

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n = 5;

```

```

// Allocate memory for an array of 5 integers using calloc
int *ptr = (int *)calloc(n, sizeof(int));

if (ptr == NULL)
{
    printf("Memory allocation failed.\n");
    return 1;
}

// Initialize and display the array elements
for (int i = 0; i < n; i++)
{
    ptr[i] = i + 1;                // Assign values
}

// Display the initialized values
printf("Array elements after calloc initialization:\n");
for (int i = 0; i < n; i++)
{
    printf("%d ", ptr[i]);
}
printf("\n");

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

```

Output:

Array elements after calloc initialization:
1 2 3 4 5

Working with calloc() in 2D Arrays:

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    int rows = 3, cols = 4;
    int **matrix = (int **)calloc(rows, sizeof(int *));

    if (matrix == NULL)
    {
        printf("Memory allocation failed.\n");
        return 1;
    }
    // Allocate memory for each row

```

```

for (int i = 0; i < rows; i++)
{
    matrix[i] = (int *)calloc(cols, sizeof(int));
    if (matrix[i] == NULL)
    {
        printf("Memory allocation for row %d failed.\n", i);
        return 1;
    }
}
// Initialize and print the matrix
printf("2D array initialized by calloc:\n");
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        printf("%d ", matrix[i][j]);           // All elements are zero
    }
    printf("\n");
}
// Free the allocated memory
for (int i = 0; i < rows; i++)
{
    free(matrix[i]);
}
free(matrix);
return 0;
}

```

Output:

2D array initialized by calloc:

```

0 0 0 0
0 0 0 0
0 0 0 0

```

- 3) **realloc()**: `realloc()` (short for **reallocation**) is a function in C used to resize a previously allocated memory block. It can increase or decrease the size of the memory block by preserving the existing data.

Syntax:

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

Example:

```

int*arr;
arr = (int*)calloc(5,sizeof(int));
.....
.....
arr=(int*)realloc(arr,sizeof(int)*10);

```

```

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

```



```

{
    int n = 3;

    // Allocate memory for 3 integers
    int *ptr = (int *)malloc(n * sizeof(int));

    if (ptr == NULL)
    {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Initialize the array
    for (int i = 0; i < n; i++)
    {
        ptr[i] = i + 1;                // Assign values: 1, 2, 3
    }

    printf("Initial array: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", ptr[i]);
    }
    printf("\n");

    // Reallocate memory for 5 integers
    n = 5;
    ptr = (int *)realloc(ptr, n * sizeof(int));

    if (ptr == NULL)
    {
        printf("Reallocation failed.\n");
        return 1;
    }

    // Initialize new elements
    for (int i = 3; i < n; i++)
    {
        ptr[i] = i + 1;                // Assign values: 4, 5
    }

    printf("Array after reallocation: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }
    printf("\n");
}

```

```
// Free the allocated memory
```

```
free(ptr);
```

```
return 0;
```

```
}
```

Output:

Initial array: 1 2 3

Array after reallocation: 1 2 3 4 5

- 4) **free()**: The free() function in C is used to deallocate or release memory that was previously allocated by dynamic memory allocation functions like malloc(), calloc(), or realloc(). When memory is no longer needed, free() is called to return it to the heap, allowing other parts of the program to use that memory.

Syntax:

```
void free(void *ptr);
```

ptr: A pointer to the memory block that needs to be freed.

Returns: free() does not return any value.

Function Pointers:

A **function pointer** in programming (primarily in languages like C and C++) is a pointer that stores the address of a function. It allows you to invoke a function indirectly through the pointer, enabling dynamic function calls and flexibility in function selection during runtime.

Syntax:

```
return_type (*pointer_name)(parameter_list);
```

Ex:

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

```
// Function definition
```

```
void displayMessage() {
```

```
    printf("Hello, Function Pointer!\n");
```

```
}
```

```
int main()
```

```
{
```

```
    // Declare a function pointer
```

```
    void (*funcPtr)();
```

```
    // Assign the address of the function
```

```
    funcPtr = displayMessage;
```

```
    // Call the function via the pointer
```

```
    funcPtr(); // Output: Hello, Function Pointer!
```

```
    return 0;
```

```
}
```

Implement Simple calculator using function pointers

```
#include <stdio.h>

// Function prototypes for basic arithmetic operations
int add(int a, int b) {
    return a + b;
}
int subtract(int a, int b) {
    return a - b;
}
int multiply(int a, int b) {
    return a * b;
}
int divide(int a, int b) {
    if (b == 0) {
        printf("Error: Division by zero!\n");
        return 0;
    }
    return a / b;
}

int main() {
    int choice, a, b;
    int (*operation)(int, int); // Function pointer

    // Display menu
    printf("Simple Calculator\n");
    printf("1. Add\n");
    printf("2. Subtract\n");
    printf("3. Multiply\n");
    printf("4. Divide\n");
    printf("Enter your choice (1-4): ");
    scanf("%d", &choice);

    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);

    // Select the function based on user choice
    switch (choice) {
        case 1:
            operation = add;
            break;
        case 2:
            operation = subtract;
            break;
        case 3:
            operation = multiply;
            break;
```

```

case 4:
    operation = divide;
    break;
default:
    printf("Invalid choice!\n");
    return 1;
}
// Call the selected function via the pointer
int result = operation(a, b);
printf("Result: %d\n", result);
return 0;
}

```

File Handling:

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program.

Why Use File Handling?

- Persistent data storage.
- Efficient handling of large datasets.
- Facilitates modular programming by separating data and logic.
- Allows data sharing between different programs.

Types of Files:

1. **Text Files:** Store data as plain text, human-readable (e.g., .txt files).
2. **Binary Files:** Store data in binary format, efficient for numerical data and structures.

File Operations in C:

C supports the following primary file operations:

1. **Creating a file:** Initialize a new file for data storage.
2. **Opening a file:** Open an existing or new file.
3. **Reading a file:** Retrieve data from the file.
4. **Writing to a file:** Add new data or overwrite existing data.
5. **Closing a file:** Free the resources associated with the file.

Functions for file handling

There are many functions in the C library to open, read, write, search and close the file.

A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file

7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

Steps to working with files:

Step-1: Declare a file pointer: It is used to store the address of a file to access.

```
FILE *fp;
```

Step-2: Opening File: fopen()

We must open a file before it can be read, write, or update. The fopen() function is used to open a file. The syntax of the fopen() is given below.

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

The fopen() function accepts two parameters:

The **file name** (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like "c://some_folder/some_file.ext".

The **mode** in which the file is to be opened. It is a string.

We can use one of the following modes in the fopen() function.

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
rb	opens a binary file in read mode
wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

Ex:

```
fp = fopen("example.txt", "w");
if (fp == NULL)
{
    printf("Error opening file.\n");
    return 1;
}
```

Step-3: Perform read/write operations

Ex:

```
fprintf(fp, "Hello, File Handling in C!");           //To write the content to a file.
```

Step-4: Close the file using fclose.

```
fclose(fp);
```

Practice Programs:

1. Write a C program to write the data to a file.

```
#include <stdio.h>
int main()
{
    FILE *fp;

    // Open file in write mode
    fp = fopen("example.txt", "w");
    if (fp == NULL)
    {
        printf("Error opening file.\n");
        return 1;
    }

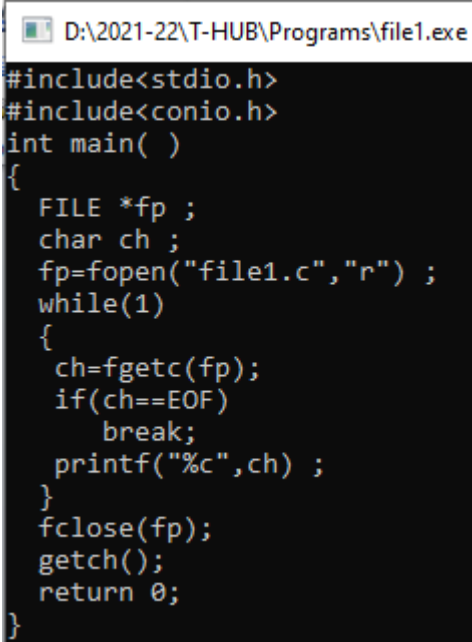
    // Write data to the file
    fprintf(fp, "Hello, File Handling in C!\n");

    //Close the file
    fclose(fp);

    printf("Data written successfully.\n");
    return 0;
}
```

2. Program to read the data from a file and display it on the screen

```
#include<stdio.h>
#include<conio.h>
int main()
{
    FILE *fp;
    char ch;
    fp=fopen("file1.c", "r");
    while(1)
    {
        ch=fgetc(fp);
        if(ch==EOF)
            break;
        printf("%c", ch);
    }
    fclose(fp);
    getch();
    return 0;
}
```



```
D:\2021-22\T-HUB\Programs\file1.exe
#include<stdio.h>
#include<conio.h>
int main( )
{
    FILE *fp ;
    char ch ;
    fp=fopen("file1.c","r") ;
    while(1)
    {
        ch=fgetc(fp);
        if(ch==EOF)
            break;
        printf("%c",ch) ;
    }
    fclose(fp);
    getch();
    return 0;
}
```

fgetc() function

The fgetc() function returns a single character from the file. It gets a character from the stream. It returns EOF at the end of file.

Syntax: int fgetc(FILE *stream)

fputc() function

The fputc() function is used to write a single character into file. It outputs a character to a stream.

Syntax: int fputc(int c, FILE *stream)


Closing File: fclose()

The fclose() function is used to close a file. The file must be closed after performing all the operations on it. The syntax of fclose() function is given below:

int fclose(FILE *fp);

3. Program to copy the data from one file to another file.

```
#include<stdio.h>
#include<conio.h>
int main()
{
    FILE *fp1,*fp2;
    char c;
    fp1=fopen("file1.c","r");
    fp2=fopen("copy.c","w");
    while ((c=fgetc(fp1))!=EOF)
    {
        fputc(c,fp2);
    }
    printf("Copied Successfully");
    fclose(fp1);
    fclose(fp2);
    getch();
    return 0;
}
```

 D:\2021-22\T-HUB\Programs\file1.exe

Copied Successfully_

Random Access File:

Random accessing of files in C language can be done with the help of the following functions –

- ftell ()
- rewind ()
- fseek ()

ftell(): The ftell() function returns the current file position of the specified stream. We can use ftell() function to get the total size of a file after moving file pointer at the end of file. We can use SEEK_END constant to move the file pointer at the end of file.

Syntax: int ftell(File *fp);

Example: int n = ftell (file pointer)

rewind(): The rewind() function sets the file pointer at the beginning of the stream. It is useful if you have to use stream many times.

Syntax: void rewind(FILE *stream)

fseek():

The fseek() function is used to set the file pointer to the specified offset. It is used to write data into file at desired location.

Syntax:

```
int fseek(FILE *stream, long int offset, int position)
```

Offset: The no of positions to be moved while reading or writing.

It can be either negative (or) positive.

Positive - forward direction.

Negative – backward direction.

Position

It can have three values, which are as follows

0 – Beginning of the file.

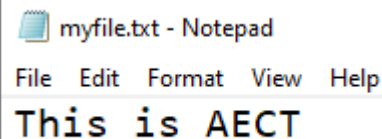
1 – Current position.

2 – End of the file.

There are 3 constants used in the fseek() function for position:

SEEK_SET, SEEK_CUR and SEEK_END.

```
#include<stdio.h>
int main()
{
    FILE *fp;
    fp = fopen("myfile.txt", "w+");
    fputs("This is ACET", fp);
    fseek( fp, 8, SEEK_SET);
    fputs("AEC", fp);
    fclose(fp);
    return 0;
}
```

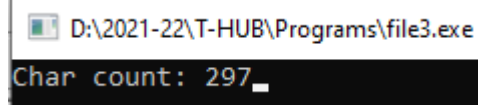


myfile.txt - Notepad
File Edit Format View Help
This is AECT

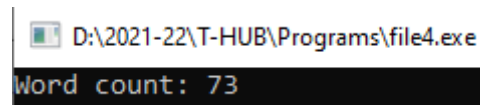
Write a C program to read a text file and count the characters and words in it?[2018]

```
#include<stdio.h>
int main()
{
    FILE *fp;
    int c=0;
    char ch;
    fp=fopen("file1.c", "r");
    while( (ch=fgetc(fp)) !=EOF)
        c++;
    printf("Char count: %d", c);
    return 0;
}

#include<stdio.h>
int main()
{
    FILE *fp;
    int w_c=0;
    char ch;
    fp=fopen("file1.c", "r");
    while( (ch=fgetc(fp)) !=EOF)
    {
        if(ch==' ' || ch=='\n')
            w_c++;
    }
    printf("Word count: %d", w_c);
    return 0;
}
```



D:\2021-22\T-HUB\Programs\file3.exe
Char count: 297



D:\2021-22\T-HUB\Programs\file4.exe
Word count: 73

Working with fread and fwrite:

fread and fwrite are standard library functions in C, used for performing **binary file I/O**. They allow efficient reading from and writing to files in chunks, making them well-suited for operations on binary files like images, audio, and video.

Syntax:

size_t fread(void *ptr, size_t size, size_t count, FILE *stream);

Where **ptr**: Pointer to the memory block where data will be stored.

size: Size of each element to be read (in bytes).

count: Number of elements to be read.

stream: File pointer to the open file.

Write a C program to copy the content from one image file to another.

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

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

```
int main() {
```

```
    FILE *sourceFile, *destFile;
```

```
    char buffer[1024];
```

```
    size_t bytesRead;
```

```
    // Open the source image file in binary read mode
```

```
    sourceFile = fopen("source.jpg", "rb");
```

```
    if (!sourceFile) {
```

```
        perror("Error opening source file");
```

```
        return 1;
```

```
    }
```

```
    // Open the destination image file in binary write mode
```

```
    destFile = fopen("copy.jpg", "wb");
```

```
    if (!destFile) {
```

```
        perror("Error opening destination file");
```

```
        fclose(sourceFile);
```

```
        return 1;
```

```
    }
```

```
    // Copy data from source to destination
```

```
    while ((bytesRead = fread(buffer, 1, sizeof(buffer), sourceFile)) > 0)
```

```
    {
```

```
        fwrite(buffer, 1, bytesRead, destFile);
```

```
    }
```

```
    // Close the files
```

```
    fclose(sourceFile);
```

```
    fclose(destFile);
```

```
    printf("Image file copied successfully.\n");
```

```
    return 0;
```

```
}
```



Source.jpg



Copy.jpg

Important Questions:

1. Define Pointer. Explain the significance of the * (dereferencing operator) and & (address-of operator) in pointers.
2. Explain how pointer arithmetic depends on the data type of the pointer.
3. Write a program to traverse an array using pointers.
4. Explain the parameter passing techniques in C. Write a C program to swap 2 numbers using call by value and address.
5. Define and differentiate between null pointers, generic pointers, and dangling pointers.
6. Differentiate between malloc, calloc, realloc, and free with examples.
7. Explain the significance of `argc` and `argv` in C. Write a program to calculate the sum of numbers passed as command-line arguments.
8. What are files in C? Differentiate between text files and binary files. Explain the modes in which a file can be opened using the `fopen` function.
9. Write a program to read data from a text file and display it on the console.
10. Write a C program to copy the content from one file to another.
11. What is random access in file handling? How is it achieved in C?
12. Write a program to demonstrate the use of `fseek` and `ftell` functions.