C-PROGRAMMING MODULE 4

1. Write a function (using pointer parameters) that compares two integer arrays to see whether they are identical. The function returns 1 if they are identical, 0 otherwise.

```
#include <stdio.h>

int compare_arrays(int *arr1, int *arr2, int size) {

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

      if (*(arr1 + i) != *(arr2 + i)) {

         return 0;  // Arrays are not identical

      }

   }

   return 1;  // Arrays are identical

}

int main() {

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

   int arr2[] = {1, 2, 3, 4, 5};

   int arr3[] = {1, 2, 3, 4, 6};

   int size = sizeof(arr1) / sizeof(arr1[0]);

   if (compare_arrays(arr1, arr2, size)) {

      printf("arr1 and arr2 are identical.\n");

   } else {

      printf("arr1 and arr2 are not identical.\n");

   }

   if (compare_arrays(arr1, arr3, size)) {

      printf("arr1 and arr3 are identical.\n");

   } else {

      printf("arr1 and arr3 are not identical.\n");

   }

   return 0;

}
```

Explanation:

- **Pointer Parameters:** `arr1` and `arr2` are pointers to the first elements of the two arrays.
- **Function Logic:** The function iterates through the arrays and compares corresponding elements using pointer arithmetic (`*(arr1 + i)`).
- **Output:** It returns `1` if the arrays are identical, and `0` otherwise.

2. What is the difference between a pointer and a normal variable? Provide examples to support your answer.

- A **normal variable** directly stores a value (such as an integer, float, or character).

- A **pointer variable** stores the **memory address** of another variable. Pointers are used for referencing and manipulating the memory location of variables.

Example:

```
#include <stdio.h>

int main() {

   int num = 10;     // normal variable

   int *ptr = &num;  // pointer variable

   printf("Value of num: %d\n", num);     // Prints 10

   printf("Address of num: %p\n", &num); // Prints the address of num

   printf("Value stored in ptr: %p\n", ptr); // Prints the address stored in ptr

   printf("Value pointed by ptr: %d\n", *ptr); // Prints the value at the address stored
in ptr, which is 10

   return 0;

}
```

- **Normal Variable (`num`)** stores a value (10).
- **Pointer (`ptr`)** stores the memory address of `num`.

Key Differences:

- Normal variable stores data directly, while a pointer stores the address of a variable.
- Pointers enable indirect access and modification of variables.

3. Explain pointer arithmetic in C with an example. Write a C program to demonstrate pointer

arithmetic (increment, decrement, addition, subtraction, and pointer difference) using an integer array. Provide a detailed explanation of the output.

Pointer arithmetic allows you to manipulate memory addresses directly using operators like +, −, ++, −−.

- **Increment (++) and Decrement (−−)**: Move the pointer to the next or previous element based on the size of the data type it points to.
- **Addition and Subtraction**: You can add or subtract an integer from a pointer. This moves the pointer by that many elements.
- **Pointer Difference**: Subtracting two pointers gives the number of elements between them.

<u>Program:</u>

```
#include <stdio.h>
int main() {
   int arr[] = {10, 20, 30, 40, 50};
   int *ptr = arr;
   // Pointer Increment
   printf("Pointer Increment: %d\n", *(ptr++)); // Prints 10, moves pointer to next element
   // Pointer Decrement
   printf("Pointer Decrement: %d\n", *(--ptr)); // Moves pointer back to previous element and prints 10
   // Pointer Addition
   ptr = arr;
   ptr = ptr + 2; // Moves pointer to 3rd element (index 2)
   printf("Pointer Addition: %d\n", *ptr); // Prints 30
   // Pointer Subtraction
   ptr = arr + 4; // Points to the last element (index 4)
   printf("Pointer Subtraction: %d\n", *(ptr - 2)); // Moves pointer back 2 positions to 3rd element (prints 30)
   // Pointer Difference
   int *ptr1 = arr;
   int *ptr2 = arr + 4;
   printf("Pointer Difference: %ld\n", ptr2 - ptr1); // Prints 4 (difference in number of elements)
   return 0;
}
```

Output:

```
Pointer Increment: 10

Pointer Decrement: 10

Pointer Addition: 30

Pointer Subtraction: 30

Pointer Difference: 4
```

Explanation:

- **Pointer Increment/Decrement:** The pointer moves based on the type it points to (e.g., moving by 1 unit of `int` size).
- **Pointer Addition/Subtraction:** You can adjust the pointer to move through an array by adding or subtracting integer values.
- **Pointer Difference:** Subtracting two pointers gives the number of elements between them, not just the raw byte difference.

4. What are the rules of pointer operations?

- A pointer can only be incremented or decremented based on the size of the type it points to.
- A pointer can be added or subtracted with an integer.
- You can subtract one pointer from another, but they must point to elements of the same array or memory block.
- Dereferencing a NULL pointer causes undefined behavior.
- You can use pointers to perform indirect access, but proper initialization is necessary before use.

5. Differentiate between char name[] and char *name in C.

- `char name[]` defines an array of characters, which is a fixed-size collection of characters stored consecutively in memory.
- `char *name` is a pointer to a character, which can point to any location in memory (e.g., a string literal or dynamically allocated memory).

Example:

```
#include <stdio.h>

int main() {

   char name[] = "Hello";   // Array of characters

   char *name_ptr = "Hello"; // Pointer to a string literal
```

```
    printf("name: %s\n", name);      // Prints Hello
    printf("name_ptr: %s\n", name_ptr); // Prints Hello
    return 0;
}
```

- **`name[]`**: The array `name[]` holds the actual characters.
- **`name_ptr`**: The pointer `name_ptr` holds the address of the string literal `"Hello"`.

6. What do you mean by a pointer variable? How is it initialized?

- A **pointer variable** is a variable that stores the **memory address** of another variable rather than storing an actual value.
- A pointer is initialized by assigning the address of a variable using the `&` operator.

Example:

```
#include <stdio.h>
int main() {
    int num = 10;
    int *ptr = &num;  // Pointer initialized with address of num
    printf("Value of num: %d\n", num);
    printf("Address of num: %p\n", &num);
    printf("Value at ptr: %d\n", *ptr);  // Dereferencing pointer
    return 0;
}
```

- **Pointer Initialization:** `int *ptr = &num;` assigns the address of `num` to `ptr`.
- **Dereferencing:** `*ptr` gives the value stored at the memory address pointed by `ptr`.

7. Write a function using pointers to add two matrices and to return the resultant matrix to the calling function.

```
#include <stdio.h>
void add_matrices(int *matrix1, int *matrix2, int *result, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            *(result + i * cols + j) = *(matrix1 + i * cols + j) + *(matrix2 + i * cols + j);
        }
    }
}
void print_matrix(int *matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", *(matrix + i * cols + j));
        }
        printf("\n");
    }
}
int main() {
    int matrix1[2][2] = {{1, 2}, {3, 4}};
    int matrix2[2][2] = {{5, 6}, {7, 8}};
    int result[2][2];
    add_matrices((int *)matrix1, (int *)matrix2, (int *)result, 2, 2);
    printf("Resultant Matrix:\n");
    print_matrix((int *)result, 2, 2);
    return 0;
}
```

Explanation:

- **Pointer Arithmetic:** The matrix elements are accessed using pointer arithmetic (e.g., `*(matrix + i * cols + j)`).
- **Functionality:** The `add_matrices` function takes two matrices (pointed to by `matrix1` and `matrix2`) and stores the result in the matrix pointed to by `result`.
- **Output:** The program adds two 2x2 matrices and prints the resultant matrix.

8. Write a program to calculate the sum of two numbers which are passed as arguments using the call by reference method.

```
#include <stdio.h>
void add_numbers(int *a, int *b, int *sum) {
    *sum = *a + *b;  // Dereference pointers to add values
}
int main() {
    int num1 = 5, num2 = 7, result;
    add_numbers(&num1, &num2, &result);  // Pass addresses of num1, num2, and result
    printf("Sum: %d\n", result);  // Prints sum
    return 0;
}
```

Explanation:

- **Call by Reference:** The addresses of the variables `num1`, `num2`, and `result` are passed to the function `add_numbers`. Inside the function, dereferencing the pointers allows the sum to be stored in `result`.
- **Functionality:** The `add_numbers` function takes pointers to `num1`, `num2`, and `sum`, calculates the sum and stores it in `sum`.

9. Explain how pointers can be passed to functions with an example.

Pointers can be passed to functions in C, which allows the function to modify the original data (since the function operates on the memory address, not a copy of the data).

Example:

```
#include <stdio.h>
void modify_value(int *x) {
    *x = 20;  // Dereference pointer to change the value of x
}
int main() {
    int num = 10;
    printf("Before: %d\n", num);
    modify_value(&num);  // Pass address of num
    printf("After: %d\n", num);  // Prints 20
    return 0;
}
```

Explanation:

- **Passing Pointers:** The address of the variable `num` is passed to the function `modify_value`. Inside the function, the pointer is dereferenced to modify the original value of `num`.
- **Output:** The value of `num` changes from 10 to 20 because the function modified the value through the pointer.

10. How can an array be passed to a function using a pointer? Provide an example.

In C, arrays are passed to functions using pointers. When you pass an array to a function, you're actually passing the address of the first element, which is treated as a pointer in the function.

Example:

```
#include <stdio.h>

void print_array(int *arr, int size) {

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

      printf("%d ", *(arr + i));  // Dereferencing pointer to print array elements

   }

   printf("\n");

}

int main() {

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

   int size = sizeof(arr) / sizeof(arr[0]);

   print_array(arr, size);  // Pass the array (as a pointer)

   return 0;

}
```

Explanation:

- **Array to Pointer:** The array `arr` is passed to the `print_array` function. In C, when an array is passed, it is treated as a pointer to its first element.
- **Pointer Arithmetic:** Inside the function, `*(arr + i)` is used to access the array elements by pointer dereferencing.
- **Output:** The program prints the array elements: `1 2 3 4 5`.

11. Differentiate between an array of pointers and a pointer to an array.

1. Array of Pointers:

An **array of pointers** is an array where each element is a pointer that can point to a different variable or memory location. It is typically used when you need to store the addresses of multiple variables (e.g., an array of strings).

Syntax:

```
type *array_name[size];
```

Example:

```
#include <stdio.h>

int main() {

   int *arr[3];  // Array of pointers (3 pointers)

   int x = 5, y = 10, z = 15;

   arr[0] = &x;  // arr[0] points to x

   arr[1] = &y;  // arr[1] points to y

   arr[2] = &z;  // arr[2] points to z

   printf("Value of x: %d\n", *arr[0]);

   printf("Value of y: %d\n", *arr[1]);

   printf("Value of z: %d\n", *arr[2]);

   return 0;

}
```

2. Pointer to an Array:

A **pointer to an array** is a pointer that points to the entire array, not just its individual elements. This is typically used when you want a single pointer to reference a whole array.

- **Syntax:**

```
type (*pointer_name)[size];
```

Example:

```
#include <stdio.h>

int main() {

   int arr[3] = {5, 10, 15};  // An array of integers

   int (*ptr)[3] = &arr;      // Pointer to an array of 3 integerS

   // Accessing array elements through pointer

   printf("First element: %d\n", (*ptr)[0]);

   printf("Second element: %d\n", (*ptr)[1]);
   printf("Third element: %d\n", (*ptr)[2]);

   return 0;

}
```

12. Explain the concept of a function to return a pointer to the calling function with example.

In C, a function can return a **pointer** to a memory location, which allows the calling function to access and manipulate the data in the location where the pointer points. This is particularly useful when working with dynamically allocated memory or arrays.

Example: Returning a Pointer to a Dynamically Allocated Memory

Here is an example of a function that returns a pointer to the calling function. The function dynamically allocates memory for an array, returns the pointer to the array, and then the calling function can use it.

```
#include <stdio.h>

#include <stdlib.h>

int* allocate_and_return_array(int size) {

   // Dynamically allocate memory for an array of integers

   int *arr = (int*) malloc(size * sizeof(int));

   // Check if memory allocation is successful

   if (arr == NULL) {

      printf("Memory allocation failed\n");

      exit(1);  // Exit if memory allocation fails

   }

   // Initialize array values

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

      arr[i] = i + 1;  // Filling array with values 1, 2, 3, ...

   }

   // Return the pointer to the dynamically allocated array

   return arr;

}

int main() {

   int size = 5;

   // Call the function and get the pointer to the dynamically allocated array

   int *array_ptr = allocate_and_return_array(size);
```

```
    // Print the array elements using the returned pointer

    printf("Array elements:\n");

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

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

    }

    printf("\n");

    // Free the allocated memory after use

    free(array_ptr);

    return 0;

}
```

Explanation:

- **Function `allocate_and_return_array`**:
  - This function dynamically allocates memory for an array of integers of the specified size using `malloc`.
  - The function initializes the array with values and returns a pointer to the array.
- **Returning the Pointer**:
  - The `return arr;` statement returns the pointer to the dynamically allocated memory (the array).
- **In `main()`**:
  - The pointer `array_ptr` receives the address of the dynamically allocated array returned by the function.
  - The program accesses and prints the array values through the pointer.

Key Points:

1. **Dynamic Memory Allocation:** The function uses `malloc` to allocate memory at runtime.
2. **Returning Pointers:** The function returns the pointer to the allocated memory (i.e., the array).
3. **Memory Management:** After using the pointer, the program calls `free()` to deallocate the memory to avoid memory leaks.

Output:

```
Array elements:

1 2 3 4 5
```

13. How do you declare and initialize a pointer to a pointer? Write a program to demonstrate this concept.

Declaring and Initializing a Pointer to a Pointer

A pointer to a pointer is a variable that holds the address of another pointer. It is often used when we need to manipulate a pointer from within a function or handle multi-level memory referencing.

Declaration:

To declare a pointer to a pointer, we use the following syntax:

```
datatype **ptr;
```

Initialization:

To initialize a pointer to a pointer, we assign it the address of an existing pointer:

```
**ptr = &ptr_to_pointer;
```

Example Program:

```
#include <stdio.h>

int main() {

   int value = 10;

   int *ptr = &value;     // Pointer to the variable 'value'

   int **ptr_to_ptr = &ptr; // Pointer to the pointer 'ptr'

   printf("Value: %d\n", value);

   printf("Using ptr: %d\n", *ptr);   // Dereferencing ptr

   printf("Using ptr_to_ptr: %d\n", **ptr_to_ptr);  // Dereferencing ptr_to_ptr twice

   return 0;

}
```

Explanation:

- `ptr` is a pointer to an integer and holds the address of `value`.
- `ptr_to_ptr` is a pointer to a pointer and holds the address of `ptr`.
- We use `*ptr` to access the value of `value` and `**ptr_to_ptr` to access the value of `value` through two levels of indirection.

14. Write a C program to reverse a string using pointers.

Program to Reverse a String Using Pointers

We can reverse a string using pointers by swapping characters from both ends of the string.

Example Program:

```
#include <stdio.h>

#include <string.h>

void reverse_string(char *str) {

  char *start = str;       // Pointer to the start of the string

  char *end = str + strlen(str) - 1; // Pointer to the end of the string

  char temp;

  while (start < end) {

    // Swap characters at start and end pointers

    temp = *start;

    *start = *end;

    *end = temp;

    // Move the pointers towards the center

    start++;

    end--;

  }

}

int main() {

  char str[] = "Hello, World!";

  printf("Original string: %s\n", str);

  reverse_string(str);  // Function call to reverse the string

  printf("Reversed string: %s\n", str);

  return 0;

}
```

Explanation:

- We use two pointers, `start` and `end`, to point to the first and last characters of the string.
- Characters at these positions are swapped, and the pointers are moved towards the center.
- This process continues until the pointers meet or cross each other.

15. Write a C program to print the elements of an array in reverse order using pointers.

```
#include <stdio.h>

int main() {

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

  int *ptr = arr + 4; // Point to the last element of the array

  printf("Array elements in reverse order:\n");

  for (int i = 4; i >= 0; i--) {

    printf("%d ", *(ptr - i)); // Access elements using pointer arithmetic

  }

  return 0;

}
```

Explanation:

- We initialize the pointer `ptr` to point to the last element of the array (`arr + 4`).
- We use pointer arithmetic (`*(ptr - i)`) to access the elements in reverse order.

16. What is an array of pointers, and how is it useful in string manipulation?

An **array of pointers** is a collection of pointers where each pointer points to a different variable or element. In string manipulation, an array of pointers is often used to store multiple strings.

Example:

```
#include <stdio.h>

int main() {

  char *strings[] = {"Hello", "World", "C", "Programming"};

  printf("Array of strings:\n");

  for (int i = 0; i < 4; i++) {

    printf("%s\n", strings[i]);  // Accessing strings via the array of pointers

  }

  return 0;

}
```

Explanation:

- `strings[]` is an array of pointers, where each element of the array points to a string literal.
- This allows efficient handling of multiple strings and is particularly useful when working with dynamic sets of strings (e.g., reading multiple words).

17. How does pointer arithmetic help in traversing a string?

Pointer arithmetic is used to navigate through the characters of a string in C. A string in C is essentially an array of characters, and pointers can be used to access each character in sequence.

Example:

```
#include <stdio.h>

int main() {

   char str[] = "Hello";

   char *ptr = str;

   // Traversing the string using pointer arithmetic

   while (*ptr != '\0') {  // Stop when we reach the null character

      printf("%c ", *ptr);

      ptr++;  // Move to the next character

   }

   return 0;

}
```

Explanation:

- `ptr++` increments the pointer to move to the next character in the string.
- This pointer arithmetic simplifies the traversal and manipulation of strings without the need for indexing.

18. Explain the difference between *ptr++ and (*ptr)++ if ptr is pointing to the first element of an array.

- **`*ptr++`**:
  - First, the `ptr` pointer is incremented, and then the value at the new address is dereferenced.
  - It is equivalent to `*(ptr++)`.
- **`(*ptr)++`**:
  - The value at the address `ptr` points to is incremented, and then `ptr` remains unchanged.
  - It is equivalent to `(*ptr)++`.

Example:

```
#include <stdio.h>

int main() {

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

   int *ptr = arr;

   printf("Using *ptr++: %d\n", *ptr++); // Dereference first, then increment

   printf("Using (*ptr)++: %d\n", (*ptr)++); // Increment value, then dereference

   printf("After operations, arr[0] = %d, arr[1] = %d\n", arr[0], arr[1]);

   return 0;

}
```

Explanation:

- The expression `*ptr++` prints the first element (`1`) and then increments the pointer to point to the next element.
- The expression `(*ptr)++` increments the value of the first element (changes `1` to `2`) but keeps the pointer pointing to the same element.

19. Write a program using pointers to compute the sum of all elements stored in an array.

```
#include <stdio.h>

int main() {

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

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

   int sum = 0, i;

   // Calculating sum using pointer

   for(i = 0; i < 5; i++) {

      sum += *(ptr + i); // Accessing elements using pointer

   }

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

   return 0;

}
```

Explanation: The pointer `ptr` is used to access each element of the array `arr` using pointer arithmetic (`*(ptr + i)`). The sum of the elements is calculated and displayed.

20. Write a function using pointers to exchange the values stored in two locations in the memory.

```
#include <stdio.h>

void exchange(int *x, int *y) {

   int temp = *x;

   *x = *y;

   *y = temp;

}

int main() {

   int a = 10, b = 20;

   printf("Before exchange: a = %d, b = %d\n", a, b);

   exchange(&a, &b); // Passing addresses of variables a and b

   printf("After exchange: a = %d, b = %d\n", a, b);

   return 0;

}
```

Explanation: The function `exchange()` takes two integer pointers as arguments, swaps the values they point to using a temporary variable `temp`. In the `main()` function, the addresses of `a` and `b` are passed to swap their values.

21. What are the advantages of using function pointer in C? Write a program that uses a function pointer as a function argument.

Advantages of Using Function Pointers:

- **Flexibility**: Function pointers allow dynamic function calls, enabling different functions to be executed based on certain conditions.
- **Callback Mechanism**: Allows passing a function as an argument to another function, useful in implementing event-driven programs or for callbacks.
- **Improved Modularization**: Function pointers enable a more modular approach in programs by decoupling function definitions from their usage.

Program Using Function Pointer as Argument:

```
#include <stdio.h>

void print_message(char *message) {

   printf("%s\n", message);

}

void execute(void (*func_ptr)(char *), char *msg) {
   func_ptr(msg); // Calling the function using the pointer
}
int main() {
   char message[] = "Hello, Function Pointer!";
   // Passing function pointer and message to execute function
   execute(print_message, message);
   return 0;
}
```

Explanation: The `execute()` function takes a function pointer as an argument and calls the passed function. The `print_message()` function prints the message passed to it.

22. Explain the difference between accessing structure members using a structure variable and a pointer to structure. Provide an example.

- **Access using Structure Variable**: Structure members are accessed directly using the `.` operator.
- **Access using Pointer to Structure**: Structure members are accessed using the `->` operator when using a pointer.

Example:

```
#include <stdio.h>
struct Student {
    int roll_no;
    char name[50];
};
int main() {
    struct Student stu = {101, "John Doe"};
    struct Student *ptr = &stu;
    // Accessing using structure variable
    printf("Using structure variable: Roll No = %d, Name = %s\n", stu.roll_no, stu.name);
    // Accessing using pointer to structure
    printf("Using pointer to structure: Roll No = %d, Name = %s\n", ptr->roll_no, ptr->name);
    return 0;
}
```

Explanation: In the first case, the structure members are accessed directly using the `.` operator, whereas in the second case, they are accessed through the pointer using the `->` operator.

23. Explain the effects of the following statements:

a) int a, *b=&a;

b) int p, *p;

c) char *s;

a) `int a, *b = &a;`

- **Explanation**: The variable `a` is declared as an integer, and `b` is declared as a pointer to integer. The pointer `b` is initialized to the address of `a`.

b) `int p, *p;`

- **Explanation**: Here, `p` is declared twice: first as an integer, and second as a pointer to integer. This is incorrect as the same name `p` is used for both a variable and a pointer. It should be corrected to use a different name for the variable and pointer.

c) `char *s;`

- **Explanation**: The variable `s` is declared as a pointer to a `char`. This means `s` will hold the address of a `char` variable.

24. What is a pointer to a structure in C? Explain with an example program how to access structure members using a pointer.

A **pointer to a structure** in C is a pointer that holds the address of a structure variable. It allows indirect access to structure members using the `->` operator.

Example Program:

```
#include <stdio.h>

struct Book {

   char title[50];

   int pages;

};

int main() {

   struct Book b1 = {"C Programming", 500};

   struct Book *ptr = &b1; // Pointer to structure

   // Accessing members using pointer

   printf("Title: %s\n", ptr->title); // Using -> to access member through pointer
```

```
    printf("Pages: %d\n", ptr->pages);

    return 0;

}
```

Explanation: The pointer `ptr` holds the address of the structure `b1`. The structure members are accessed using the `->` operator, which allows indirect access to `title` and `pages`.

25. Explain the purpose and working of malloc(), calloc(), realloc(), and free() functions in C with suitable examples.

Dynamic memory allocation in C is used to allocate memory at runtime. The standard library functions for memory allocation and deallocation are:

1. **`malloc()` (Memory Allocation):**
   - **Purpose**: Allocates a specified number of bytes in memory and returns a pointer to the allocated space.
   - **Working**: It takes the size in bytes as an argument and returns a pointer to the first byte of the allocated memory block.
   - **Example**:

```
int *ptr;

ptr = (int *)malloc(5 * sizeof(int)); // Allocates memory for 5 integers

if (ptr == NULL) {

    printf("Memory allocation failed!\n");

}
```

2. **`calloc()` (Contiguous Allocation):**

- **Purpose**: Allocates memory for an array of elements, initializes each element to zero.
- **Working**: Takes two arguments: the number of elements and the size of each element. It returns a pointer to the allocated block, initialized to zero.
- **Example**:

```
int *ptr;

ptr = (int *)calloc(5, sizeof(int)); // Allocates and initializes 5 integers to 0

if (ptr == NULL) {

    printf("Memory allocation failed!\n");

}
```

3. **`realloc()` (Reallocation):**

- **Purpose**: Resizes a previously allocated memory block to a new size.
- **Working**: It takes two arguments: the pointer to the previously allocated block and the new size. It may either shrink or expand the block.
- **Example**:

```
ptr = (int *)realloc(ptr, 10 * sizeof(int)); // Resizes memory to hold 10 integers

if (ptr == NULL) {

   printf("Memory reallocation failed!\n");

}
```

4. **`free()` (Deallocate):**

- **Purpose**: Frees the memory previously allocated by `malloc()`, `calloc()`, or `realloc()`.
- **Working**: It takes the pointer to the memory block as an argument and releases the memory back to the system.
- **Example**:

  ```
  free(ptr); // Frees the memory allocated to ptr

  ptr = NULL; // Optional, to avoid dangling pointer
  ```

26. What is dynamic memory allocation in C? How does it differ from static memory allocation?

Explain with an example.

Dynamic memory allocation allows programs to request memory during runtime rather than at compile time. This means that memory is allocated as needed, which is more flexible and efficient than static memory allocation.

Difference from Static Memory Allocation:

- **Static Memory Allocation**: Memory is allocated at compile time, and the size must be known in advance. The memory cannot be resized during execution. E.g., arrays with fixed sizes.
- **Dynamic Memory Allocation**: Memory is allocated during runtime using functions like `malloc()`, `calloc()`, `realloc()`, and is freed using `free()`.

Example:

```
// Static allocation

int arr[5]; // Fixed size array

// Dynamic allocation

int *arr = (int *)malloc(5 * sizeof(int)); // Size allocated at runtime
```

27. Write a C program to dynamically allocate memory for an array of integers, take input values from the user, and print the array. Use malloc() for allocation.

```
#include <stdio.h>

#include <stdlib.h>

int main() {

   int *arr, n, i;

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

   scanf("%d", &n);

   // Dynamically allocating memory

   arr = (int *)malloc(n * sizeof(int));

   if (arr == NULL) {

      printf("Memory allocation failed!\n");

      return 1;

   }

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

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

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

   }

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

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

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

   }

   // Free the allocated memory

   free(arr);

   return 0;

}
```

28. What are the consequences of not using free() after dynamic memory allocation in C?

Consequences of Not Using `free()` After Dynamic Memory Allocation

If `free()` is not called after using `malloc()`, `calloc()`, or `realloc()`, it results in **memory leaks**. The allocated memory remains reserved and cannot be reused, causing the program to consume unnecessary memory, which can eventually lead to system instability or crashes, especially in long-running programs or those with frequent memory allocations.

29. What are the differences between malloc() and calloc() in C?

Feature	malloc()	calloc()
Memory Init	Does not initialize the allocated memory (it contains garbage values).	Initializes all allocated memory to zero.
Arguments	Takes a single argument (size in bytes).	Takes two arguments (number of elements and size of each element).
Syntax Example	ptr = malloc(10 * sizeof(int));	ptr = calloc(10, sizeof(int));

30. What is dynamic memory allocation? Why is it needed?

Dynamic memory allocation refers to the process of allocating memory at runtime, allowing the size of memory blocks to be determined based on user input or program needs, unlike static allocation which requires predefined sizes.

Why is it Needed?

- **Flexibility**: Memory can be allocated as needed and can be resized (with `realloc()`).
- **Efficiency**: It helps in managing memory more efficiently, particularly for large data structures.
- **Avoids wastage**: It avoids the wastage of memory when the size is unknown or variable during runtime.

31. Explain the different modes of opening a file with an example.

In C, files can be opened using the `fopen()` function, which allows access to a file in different modes. The mode determines how the file is accessed, whether for reading, writing, or appending.

File Modes in C:

1. **`"r"` – Read Mode**:
    - Opens a file for reading. The file must exist; otherwise, the program will return `NULL`.
    - **Example**:

    ```
    FILE *file = fopen("example.txt", "r");
    if (file == NULL) {
        printf("File not found\n");
    }
    ```

2. **"w" – Write Mode**:
    - Opens a file for writing. If the file exists, it truncates the file to zero length (i.e., deletes the content). If the file does not exist, it creates a new file.
    - **Example**:

```
FILE *file = fopen("example.txt", "w");
if (file == NULL) {
    printf("Failed to open file\n");
}
```

3. **"a" – Append Mode**:
    - Opens a file for appending. If the file exists, data is written at the end of the file without truncating it. If the file does not exist, it is created.
    - **Example**:

```
FILE *file = fopen("example.txt", "a");
if (file == NULL) {
    printf("Failed to open file\n");
}
```

4. **"r+" – Read and Write Mode**:
    - Opens a file for both reading and writing. The file must exist; otherwise, it will return `NULL`.
    - **Example**:

```
FILE *file = fopen("example.txt", "r+");
if (file == NULL) {
    printf("File not found\n");
}
```

5. **"w+" – Write and Read Mode**:
    - Opens a file for both reading and writing. If the file exists, it truncates the file to zero length. If the file does not exist, it creates a new file.
    - **Example**:

```
FILE *file = fopen("example.txt", "w+");
if (file == NULL) {
    printf("Failed to open file\n");
}
```

6. **"a+" – Append and Read Mode**:
    - Opens a file for both appending and reading. If the file exists, data is appended at the end without truncating. If the file does not exist, it is created.
    - **Example**:

```
FILE *file = fopen("example.txt", "a+");
if (file == NULL) {
    printf("Failed to open file\n");
}
```

7. **"b" – Binary Mode**:
    - Used in combination with other modes to specify binary files. For example, `"rb"` or `"wb"` to open a binary file for reading or writing.

- **Example**:

  ```
  FILE *file = fopen("example.bin", "rb");
  if (file == NULL) {
      printf("Failed to open file\n");
  }
  ```

32. Explain any 5 file handling functions in C.

Here are five commonly used file handling functions in C:

1. **`fopen()`:**
   - Used to open a file. It returns a pointer to the file.
   - **Syntax**: `FILE *fopen(const char *filename, const char *mode);`
   - **Example**:

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

2. **`fclose()`:**

- Used to close a file. After closing a file, any further file operations on the file pointer will result in an error.
- **Syntax**: `int fclose(FILE *stream);`
- **Example**:

  ```
  fclose(file);
  ```

3. **`fscanf()`:**

- Reads formatted data from a file.
- **Syntax**: `int fscanf(FILE *stream, const char *format, ...);`
- **Example**:

  ```
  int x;

  fscanf(file, "%d", &x);
  ```

4. **`fscanf()`:**

- Writes formatted data to a file.
- **Syntax**: `int fprintf(FILE *stream, const char *format, ...);`
- **Example**:

  ```
  fprintf(file, "The value is %d", x);
  ```

5. **`fgetc()`:**

- Reads a single character from a file.
- **Syntax**: `int fgetc(FILE *stream);`

- **Example**:

```
char ch = fgetc(file);
```

33. Write a program to read data from the keyboard, write it to a file called INPUT, again read the same data from the INPUT file, and display it on the screen.

The following program reads data from the user, writes it to a file, reads the same data from the file, and displays it on the screen.

Example Program:

```
#include <stdio.h>
int main() {
   FILE *file;
   char str[100];
   // Read data from keyboard
   printf("Enter a string: ");
   fgets(str, sizeof(str), stdin);
   // Write data to a file
   file = fopen("INPUT.txt", "w");
   if (file == NULL) {
      printf("Unable to open the file for writing\n");
      return 1;
   }
   fprintf(file, "%s", str);
   fclose(file);  // Close the file after writing
   // Read data from the file
   file = fopen("INPUT.txt", "r");
   if (file == NULL) {
      printf("Unable to open the file for reading\n");
      return 1;
   }
   printf("Data read from file: ");
```

```
    fgets(str, sizeof(str), file);

    printf("%s", str);

    fclose(file);  // Close the file after reading

    return 0;

}
```

Explanation:

- The program first reads a string from the user and writes it to a file `INPUT.txt`.
- Then, it reads the content of the file and displays it.

34. How can you perform read and write operations on an unformatted data file in C?

Performing Read and Write Operations on an Unformatted Data File in C

Unformatted data files store data in binary form, as opposed to text files, which store data in a human-readable form. In C, we can use `fread()` and `fwrite()` to handle unformatted data.

Example:

```
#include <stdio.h>

int main() {

    FILE *file;

    int data = 123;

    // Write unformatted data to a file

    file = fopen("data.bin", "wb");

    if (file == NULL) {

        printf("Error opening file\n");

        return 1;

    }

    fwrite(&data, sizeof(int), 1, file);

    fclose(file);

    // Read unformatted data from the file

    file = fopen("data.bin", "rb");

    if (file == NULL) {

        printf("Error opening file\n");

        return 1;
    }
    fread(&data, sizeof(int), 1, file);
    printf("Data read from file: %d\n", data);
    fclose(file);
    return 0;
}
```

Explanation:

- `fwrite()` is used to write binary data, and `fread()` is used to read binary data from a file.
- Both functions use the size of the data being written or read.

35. When a program is terminated, all the files used by it are automatically closed. Why is it then necessary to close a file during execution of the program?

While files are automatically closed when a program terminates, explicitly closing files during execution is important for the following reasons:

1. **Resource Management**:
   - File descriptors (handles) are finite. If a file is not closed properly, it can lead to resource leakage (i.e., file handles are not released).
2. **Data Integrity**:
   - When writing data to a file, it may not be immediately written to disk. The data may be buffered in memory. Calling `fclose()` ensures that all buffered data is flushed to the file and the file is properly closed.
3. **Avoiding File Corruption**:
   - If a file is not closed explicitly, it may result in file corruption, especially if the program crashes or ends abruptly.

Example:

```
FILE *file = fopen("file.txt", "w");
// Operations on the file
fclose(file);  // Ensure data is written and file is closed
```

36. Write any three file-handling functions in C.

Here are three file-handling functions commonly used in C:

1. **`fopen()`**: Opens a file.
2. **`fclose()`**: Closes a file.
3. **`fgetc()`**: Reads a single character from a file.

Each of these functions plays a critical role in opening, reading, writing, and closing files efficiently in C.

37. Write a program in C to copy the contents of one file into another.

The following program demonstrates how to copy the contents of one file into another using file handling functions in C:

```
#include <stdio.h>

int main() {

  FILE *sourceFile, *destinationFile;

  char ch;

  // Open the source file in read mode

  sourceFile = fopen("source.txt", "r");

  if (sourceFile == NULL) {

    printf("Source file not found.\n");

    return 1;

  }

  // Open the destination file in write mode

  destinationFile = fopen("destination.txt", "w");

  if (destinationFile == NULL) {

    printf("Unable to open destination file.\n");

    fclose(sourceFile);

    return 1;

  }

  // Copy content from source file to destination file

  while ((ch = fgetc(sourceFile)) != EOF) {

    fputc(ch, destinationFile);

  }

  // Close both files

  fclose(sourceFile);

  fclose(destinationFile);

  printf("File copied successfully.\n");

  return 0;
```

}

Explanation:

- `fopen()`: Opens the source file in read mode (`"r"`) and the destination file in write mode (`"w"`).
- `fgetc()`: Reads one character from the source file.
- `fputc()`: Writes the character to the destination file.
- `fclose()`: Closes both files once the operation is comp

38. Write a C program to count the number of lines in a text file. The program should: Open a file in read mode using fopen(), read characters using fgetc(), and count newline (\n) characters; Display the total number of lines and close the file using fclose().

The following program counts the number of lines in a text file by reading each character using `fgetc()` and counting newline characters (`'\n'`).

```
#include <stdio.h>

int main() {

   FILE *file;

   char ch;

   int lineCount = 0;

   // Open the file in read mode

   file = fopen("textfile.txt", "r");

   if (file == NULL) {

      printf("Unable to open the file.\n");

      return 1;

   }

   // Read each character from the file and count newlines

   while ((ch = fgetc(file)) != EOF) {

      if (ch == '\n') {

         lineCount++;

      }

   }

   // Close the file

   fclose(file);

   // Output the total number of lines

   printf("Total number of lines: %d\n", lineCount);
```

```
    return 0;

}
```

Explanation:

- `fgetc()`: Reads each character from the file.
- `if (ch == '\n')`: Checks for newline characters, indicating the end of a line.
- `fclose()`: Closes the file after reading.

39. Write a C program to replace vowels in a text file with the character ‘x’.

This program reads a text file, replaces all vowels (both uppercase and lowercase) with `'x'`, and writes the updated content back to the file.

```
#include <stdio.h>

#include <ctype.h>

int main() {

    FILE *file;

    char ch;

    // Open the file in read and write mode

    file = fopen("textfile.txt", "r+");

    if (file == NULL) {

        printf("Unable to open the file.\n");

        return 1;

    }

    // Read and replace vowels

    while ((ch = fgetc(file)) != EOF) {

        if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u' ||

            ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U') {

            fseek(file, -1, SEEK_CUR);  // Move the pointer back to replace the vowel

            fputc('x', file);

        }

    }


    // Close the file

    fclose(file);
    printf("Vowels replaced with 'x' successfully.\n");
    return 0;
}
```

Explanation:

- **`fgetc()`**: Reads each character from the file.
- **`fseek()`**: Moves the file pointer back by one position to overwrite the character (vowel) with `'x'`.
- **`fputc()`**: Writes `'x'` to the file at the current position.

40. Explain the use of fseek() and ftell() in file handling.

1. **`fseek()`:**

- The `fseek()` function is used to set the file pointer to a specific location in a file.
- Syntax: `int fseek(FILE *stream, long int offset, int whence);`
  - `stream`: Pointer to the file.
  - `offset`: The number of bytes to move the file pointer.
  - `whence`: The reference point, which can be one of `SEEK_SET` (beginning), `SEEK_CUR` (current position), or `SEEK_END` (end of the file).
- **Example:**

```
FILE *file = fopen("file.txt", "r");
fseek(file, 10, SEEK_SET);  // Move the pointer to the 10th byte from the beginning
```

2. **`ftell()`:**

- The `ftell()` function returns the current position of the file pointer.
- Syntax: `long int ftell(FILE *stream);`
- **Example:**

```
long position = ftell(file);
printf("Current file pointer position: %ld\n", position);
```

41. What is the difference between fgets() and fscanf() for reading a file?

1. **`fgets()`:**

- Reads an entire line from the file, including spaces, until a newline character (`'\n'`) or the end of the file (`EOF`) is reached.
- **Syntax**: `char *fgets(char *str, int n, FILE *stream);`
- It is useful when you need to read a whole line, including spaces.
- **Example:**

```
char str[100];
```

```
fgets(str, sizeof(str), file);
```

2. **`fscanf()`:**

- Reads formatted input from the file, similar to `scanf()`, but you can specify a format string to extract specific data (e.g., integers, strings).
- **Syntax**: `int fscanf(FILE *stream, const char *format, ...);`
- It is useful for reading formatted data like numbers, strings, etc.
- **Example:**

```
int num;

fscanf(file, "%d", &num);
```

42. What is the purpose of the FILE pointer in C?

In C, the `FILE` pointer is used to represent an open file stream. It is defined in the `stdio.h` header file. A `FILE` pointer is required to perform operations such as reading, writing, and manipulating files.

- **Purpose:**
  - The `FILE` pointer allows functions like `fopen()`, `fclose()`, `fgetc()`, `fputc()`, and many others to access and perform operations on files.
  - **Example:**

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

if (file != NULL) {

  // Perform file operations

  fclose(file);

}
```

The `FILE` pointer is essential for file handling in C, enabling the system to track the current file and position for read and write operations.

43. Write a C program to open a text file, move the file pointer to a specific position using fseek(), and display the content from that position.

This program demonstrates how to open a file, move the file pointer to a specific position using `fseek()`, and display the content from that position:

```
#include <stdio.h>

int main() {

  FILE *file;

  char ch;

  long position;

  // Open the file in read mode
```

```
    file = fopen("sample.txt", "r");
    if (file == NULL) {
        printf("Unable to open the file.\n");
        return 1;
    }
    // Move the file pointer to a specific position
    position = 10;  // Move to the 10th byte in the file
    fseek(file, position, SEEK_SET);
    // Display the content from the new position
    while ((ch = fgetc(file)) != EOF) {
        putchar(ch);  // Print the character to the screen
    }
    // Close the file
    fclose(file);
    return 0;
}
```

Explanation:

- **`fseek(file, position, SEEK_SET)`**: Moves the file pointer to the 10th byte from the beginning (`SEEK_SET`).
- **`fgetc()`**: Reads one character at a time from the file, starting from the new position.
- **`putchar(ch)`**: Displays the character to the screen.

44. Write a program to open a file named INVENTORY and store in it the following data:

Item name	Number	Price	Quantity
AAA-1	111	17.50	115
BBB-2	125	36.00	75
C-3	247	31.75	104

Extend the program to read this data from the file INVENTORY and display the inventory table with the value of each item.

This program demonstrates how to store inventory data in a file, then read and display it, calculating the value for each item.

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

```c
struct Inventory {
    char name[20];
    int number;
    float price;
    int quantity;
};
int main() {
    FILE *file;
    struct Inventory item;
    // Open the file in write mode to store data
    file = fopen("INVENTORY.txt", "w");
    if (file == NULL) {
        printf("Unable to open the file.\n");
        return 1;
    }
    // Write inventory data to the file
    fprintf(file, "%s %d %.2f %d\n", "AAA-1", 111, 17.50, 115);
    fprintf(file, "%s %d %.2f %d\n", "BBB-2", 125, 36.00, 75);
    fprintf(file, "%s %d %.2f %d\n", "C-3", 247, 31.75, 104);
    fclose(file);
    // Re-open the file in read mode to display the data
    file = fopen("INVENTORY.txt", "r");
    if (file == NULL) {
        printf("Unable to open the file for reading.\n");
        return 1;
    }
    // Display inventory data with value calculation
    printf("Item Name   Number   Price   Quantity   Value\n");
    while (fscanf(file, "%s %d %f %d", item.name, &item.number, &item.price, &item.quantity) !=
EOF) {
        float value = item.price * item.quantity;
    printf("%s  %d  %.2f  %d  %.2f\n", item.name, item.number, item.price, item.quantity,
value);
  }
  fclose(file);
  return 0;
}
```

Explanation:

- **`fprintf()`**: Writes data to the file in a formatted manner.
- **`fscanf()`**: Reads data from the file into the `Inventory` structure.
- **Value calculation**: The value of each item is calculated by multiplying price and quantity.

45. Write a C program which computes the factorial of a given number and write the result to a file named factorial.

```
#include<stdio.h>
int main()
{
 int n,i,f;
 FILE *fp;
 fp=fopen("Factorial.txt","w");
 printf("Enter the number\n");
 scanf("%d",&n);
 if(n==0)
 {
 fprintf(fp,"Factorial is 1\n");
 }
else if(n<0)
 {
 fprintf(fp,"Factorial does not exist\n");
 }
 else
 {
 f=1;
 for(i=n;i>=1;i--)
 {
 f=f*i;
 }
 fprintf(fp,"Factorial=%d\n",f);
 }
 fclose(fp);
 return 0;
}
```

46. Is it possible to read from and write to the same file without resetting the file pointer? Justify your answer.

It is **possible** to read from and write to the same file **without resetting the file pointer**, but it depends on the mode in which the file is opened.

- **Modes**: To both read and write to the same file, you should open the file in `"r+"`, `"w+"`, or `"a+"` mode, which allows both reading and writing.
- **Issue**: The problem is that the file pointer may move during reading or writing operations, causing the file pointer to be out of sync for the next operation if not **managed properly.**

For example:

- After reading from a file, if you want to write, the pointer might still be at the point where reading ended. You need to use `fseek()` to move the pointer if required.

47. Explain the purpose of ftell(fp) in C. If ftell(fp) returns 50, what does this indicate?

- **Purpose of `ftell(fp)`**:
    - The `ftell()` function returns the current file pointer position in the file stream, which is the byte offset from the beginning of the file.
    - It is useful for tracking the current position in a file, especially when you are performing operations like seeking or reading from specific locations.
- **If `ftell(fp)` returns 50**:
    - It indicates that the file pointer is currently 50 bytes from the beginning of the file.

48. Explain the purpose and usage of the fseek() and fwrite() functions in file handling.

- **`fseek()`:**

- **Purpose**: The `fseek()` function is used to move the file pointer to a specific location within a file. This is useful for random access operations.
- **Syntax**: `int fseek(FILE *stream, long offset, int whence);`

- `stream`: File pointer.
- `offset`: Number of bytes to move the pointer.
- `whence`: Reference point (`SEEK_SET`, `SEEK_CUR`, `SEEK_END`).

- **`fwrite()`:**

- **Purpose**: The `fwrite()` function is used to write data to a file. It is typically used to write blocks of data (e.g., structs, arrays) to a file.
- **Syntax**: `size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);`
  - `ptr`: Pointer to the data to be written.
  - `size`: Size of each element to be written.
  - `count`: Number of elements to be written.
  - `stream`: File pointer.