MODUL2:- Introduction to Programming

THEORY EXERCISE:

Q1:- Write an essay covering the history and evolution of C programming. Explain its importance and why it is still used today.

The History and Evolution of C Programming

C programming language, developed by Dennis Ritchie at Bell Labs in the early 1970s, emerged as a successor to the B language and was designed to efficiently implement the Unix operating system. Its ability to provide low-level access to memory made it ideal for system software and performance-critical applications.

Key Milestones:

- **Standardization**: C was standardized in 1989 (ANSI C), ensuring portability and consistency across platforms. Subsequent standards, C99 and C11, introduced features like inline functions and multithreading support.
- **Influence**: C has influenced many modern languages, including C++, Java, and C#. Its principles of structured programming and efficiency have become foundational in software development.

Importance Today:

1. **Performance**: C allows for highly efficient code, crucial for systems programming and applications requiring speed.
2. **Portability**: C code can be compiled on various platforms with minimal changes, making it suitable for cross-platform development.
3. **Legacy Systems**: Many existing systems are written in C, necessitating ongoing use and maintenance.
4. **Educational Value**: C is often taught as a foundational language, helping students understand core programming concepts and memory management.
5. **Robust Ecosystem**: A vast array of libraries and tools supports C, enhancing developer productivity.

In summary, C's historical significance, ongoing relevance, and foundational role in programming education and system development underscore its lasting impact on the field of computer science.

Q2:- Describe the steps to install a C compiler (e.g., GCC) and set up an Integrated Development Environment (IDE) like DevC++, VS Code, or CodeBlocks.

Step 1: Install GCC Compiler

1. **Download MinGW**:
    - Go to the MinGW-w64 website.
    - Download the installer (e.g., **mingw-w64-install.exe**).
2. **Run the Installer**:
    - Choose the architecture (e.g., x86_64 for 64-bit).
    - Select the threads model (e.g., posix) and exception model (e.g., seh).
    - Choose the installation directory (e.g., **C:\mingw-w64**).
3. **Add to System Path**:
    - Right-click on "This PC" or "My Computer" and select "Properties."
    - Click on "Advanced system settings" and then "Environment Variables."
    - Under "System variables," find the **Path** variable and click "Edit."
    - Add the path to the **bin** directory of MinGW (e.g., **C:\mingw-w64\bin**).
4. **Verify Installation**:
    - Open Command Prompt and type **gcc --version**. If installed correctly, it will display the GCC version.
    -

Step 2: Set Up an Integrated Development Environment (IDE)

Option 1: DevC++ (Windows)

1. **Download DevC++**:
    - Go to the Dev-C++ website.
    - Download the latest version.
2. **Install DevC++**:
    - Run the installer and follow the prompts to complete the installation.
3. **Configure Compiler**:
    - Open DevC++.
    - Go to "Tools" > "Compiler Options."
    - Ensure that the selected compiler is GCC.

4. **Create a New Project**:
    - Click on "File" > "New" > "Project" to start coding.

Q3:- Explain the basic structure of a C program, including headers, main function, comments, data types, and variables. Provide examples.

A C program consists of several key components that work together to define its functionality. Below is an explanation of the basic structure, including headers, the main function, comments, data types, and variables, along with examples.

1. Headers

Headers are files that contain declarations of functions and macros. They are included at the beginning of a C program using the **#include** preprocessor directive. Common headers include:

- **<stdio.h>**: Standard Input/Output library for functions like **printf** and **scanf**.
- **<stdlib.h>**: Standard library for functions like memory allocation and random number generation.

Example:

```
#include <stdio.h>

#include <stdlib.h>
```

2. The Main Function

The **main** function is the entry point of every C program. It is where the execution starts. The function can return an integer value, typically **0** to indicate successful execution.

Example:

```
int main() {

   // Code goes here

   return 0;

}
```

3. Comments

Comments are used to explain the code and make it more readable. They are ignored by the compiler. In C, comments can be single-line or multi-line.

- **Single-line comments** start with **//**.
- **Multi-line comments** are enclosed between **/*** and ***/**.

Example:

```
// This is a single-line comment
/*
This is a multi-line comment
that spans multiple lines.
*/
```

4. Data Types

C supports several basic data types, which define the type of data a variable can hold. Common data types include:

- **int**: Integer type.
- **float**: Floating-point type for decimal numbers.
- **double**: Double-precision floating-point type.
- **char**: Character type for single characters.

Example:

```
#include <stdio.h>  // Include standard I/O header

// Main function - entry point of the program
int main() {
   // Variable declarations
   int age = 25;             // Integer variable
   float salary = 50000.50;    // Float variable
   char grade = 'A';          // Character variable

   // Print the values of the variables
   printf("Age: %d\n", age);              // %d is used for integers
   printf("Salary: %.2f\n", salary);       // %.2f is used for floats
   printf("Grade: %c\n", grade);           // %c is used for characters

   return 0;  // Return 0 to indicate successful execution
}
```

5. Variables

Variables are used to store data that can be modified during program execution. Each variable must be declared with a specific data type before it can be used.

Example:

```
int age;         // Integer type
float salary;    // Floating-point type
double pi;       // Double-precision type
char grade;      // Character type
```

Q4:- Write notes explaining each type of operator in C: arithmetic, relational, logical, assignment, increment/decrement, bitwise, and conditional operators.

C programming language provides a variety of operators that allow developers to perform operations on variables and values. Below are the main types of operators in C, along with explanations and examples for each.

1. Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations.

- **Addition (+)**: Adds two operands.
- **Subtraction (-)**: Subtracts the second operand from the first.
- **Multiplication (*)**: Multiplies two operands.
- **Division (/)**: Divides the first operand by the second (integer division if both operands are integers).
- **Modulus (%)**: Returns the remainder of the division of the first operand by the second.

Example:

```
int a = 10, b = 3;
int sum = a + b;       // sum = 13
int difference = a - b; // difference = 7
int product = a * b;   // product = 30
int quotient = a / b;  // quotient = 3
int remainder = a % b;  // remainder = 1
```

2. Relational Operators

Relational operators are used to compare two values. They return a boolean value (**true** or **false**).

- **Equal to (==)**: Checks if two operands are equal.
- **Not equal to (!=)**: Checks if two operands are not equal.
- **Greater than (>)**: Checks if the left operand is greater than the right.
- **Less than (<)**: Checks if the left operand is less than the right.
- **Greater than or equal to (>=)**: Checks if the left operand is greater than or equal to the right.
- **Less than or equal to (<=)**: Checks if the left operand is less than or equal to the right.

Example:

```
int x = 5, y = 10;

bool isEqual = (x == y);      // isEqual = false

bool isNotEqual = (x != y);    // isNotEqual = true

bool isGreater = (x > y);      // isGreater = false

bool isLess = (x < y);         // isLess = true
```

3. Logical Operators

Logical operators are used to combine multiple boolean expressions.

- **Logical AND (&&)**: Returns true if both operands are true.
- **Logical OR (||)**: Returns true if at least one of the operands is true.
- **Logical NOT (!)**: Reverses the boolean value of the operand.

```
bool a = true, b = false;

bool result1 = a && b; // result1 = false

bool result2 = a || b; // result2 = true

bool result3 = !a;    // result3 = false
```

4. Assignment Operators

Assignment operators are used to assign values to variables. The most common assignment operator is =.

- **Simple assignment (=)**: Assigns the right operand to the left operand.
- **Add and assign (+=)**: Adds the right operand to the left operand and assigns the result to the left operand.
- **Subtract and assign (-=)**: Subtracts the right operand from the left operand and assigns the result to the left operand.
- **Multiply and assign (*=)**: Multiplies the left operand by the right operand and assigns the result to the left operand.
- **Divide and assign (/=)**: Divides the left operand by the right operand and assigns the result to the left operand.
- **Modulus and assign (%=)**: Takes the modulus using two operands and assigns the result to the left operand.

Example:

```
int a = 5;

a += 3; // a = 8

a -= 2; // a = 6

a *= 2; // a = 12

a /= 3; // a = 4

a %= 3; // a = 1
```

5. Increment/Decrement Operators

Increment and decrement operators are used to increase or decrease the value of a variable by one.

- **Increment (++)**: Increases the value of a variable by 1. It can be used in two forms:
    - **Prefix (++x)**: Increments the value before it is used in an expression.
    - **Postfix (x++)**: Increments the value after it is used in an expression.
- **Decrement (--)**: Decreases the value of a variable by 1. It can also be used in two forms:
    - **Prefix (--x)**: Decrements the value before it is used in an expression.
    - **Postfix (x--)**: Decrements the value after it is used in an expression.

Example:

```
int x = 5;

int y = ++x; // y = 6, x = 6 (prefix)
```

int z = x--; // z = 6, x = 5 (postfix)

6. Bitwise Operators

Bitwise operators perform operations on the binary representations of integers.

- **Bitwise AND (&)**: Compares each bit of two operands and returns a new integer with bits set to 1 where both operands have bits set to 1.
- **Bitwise OR (|)**: Compares each bit of two operands and returns a new integer with bits set to 1 where at least one operand has bits set to 1.
- **Bitwise XOR (^)**: Compares each bit of two operands and returns a new integer with bits set to 1 where the bits of the operands are different.
- **Bitwise NOT (~)**: Inverts the bits of the operand.
- **Left Shift (<<)**: Shifts the bits of the left operand to the left by the number of positions specified by the right operand.
- **Right Shift (>>)**: Shifts the bits of the left operand to the right by the number of positions specified by the right operand.

Example:

```
int a = 5;  // Binary: 0101

int b = 3;  // Binary: 0011

int andResult = a & b; // andResult = 1 (Binary: 0001)

int orResult = a | b;  // orResult = 7 (Binary: 0111)

int xorResult = a ^ b; // xorResult = 6 (Binary: 0110)

int notResult = ~a;    // notResult = -6 (Binary: 1010 in two's complement)

int leftShift = a << 1; // leftShift = 10 (Binary: 1010)

int rightShift = a >> 1; // rightShift = 2 (Binary: 0010)
```

7. Conditional (Ternary) Operator

The conditional operator (also known as the ternary operator) is a shorthand for the **if-else** statement. It takes three operands and returns one of two values based on a condition.

Syntax:

c

Run code

```
condition ? expression1 : expression2;
```

If the condition is true, **expression1** is evaluated; otherwise, **expression2** is evaluated.

Example:

```
int a = 10, b = 20;

int max = (a > b) ? a : b; // max = 20
```

Q5:- Explain decision-making statements in C (if, else, nested if-else, switch). Provide examples of each.

Decision-making statements in C allow the program to execute different actions based on certain conditions. The primary decision-making statements are **if**, **else**, **nested if-else**, and **switch**. Below is an explanation of each, along with examples.

1. if Statement

The **if** statement evaluates a condition and executes a block of code if the condition is true.

Syntax:

```
if (condition) {

   // Code to execute if condition is true

}
```

Example:

```
#include <stdio.h>


int main() {

   int number = 10;


   if (number > 0) {

      printf("The number is positive.\n");

   }


   return 0;

}
```

Output: The number is positive.

2. else Statement

The **else** statement can be used in conjunction with the **if** statement to execute a block of code when the condition is false.

Syntax:

```
if (condition) {
  // Code to execute if condition is true
} else {
  // Code to execute if condition is false
}
```

Example:

```
#include <stdio.h>

int main() {
  int number = -5;

  if (number > 0) {
    printf("The number is positive.\n");
  } else {
    printf("The number is not positive.\n");
  }

  return 0;
}
```

Output: The number is not positive.

3. Nested if-else Statement

A nested **if-else** statement is an **if** statement inside another **if** statement. This allows for more complex decision-making.

```
if (condition1) {
  // Code to execute if condition1 is true
  if (condition2) {
      // Code to execute if condition2 is true
  } else {
      // Code to execute if condition2 is false
  }
} else {
  // Code to execute if condition1 is false
}
```

Example:

```
#include <stdio.h>

int main() {
    int number = 0;

    if (number > 0) {
        printf("The number is positive.\n");
    } else if (number < 0) {
        printf("The number is negative.\n");
    } else {
        printf("The number is zero.\n");
    }

    return 0;
}
```

Output: The number is zero.

4. switch Statement

The **switch** statement allows for multi-way branching based on the value of a variable. It evaluates an expression and executes the corresponding case block.

Syntax:

```
switch (expression) {
    case constant1:
        // Code to execute if expression equals constant1
        break;
    case constant2:
        // Code to execute if expression equals constant2
        break;
    // ...
    default:
        // Code to execute if no case matches
}
```

Example:

```
#include <stdio.h>

int main() {
    int day = 3;

    switch (day) {
        case 1:
            printf("Monday\n");
            break;
        case 2:
            printf("Tuesday\n");
            break;
        case 3:
            printf("Wednesday\n");
            break;
        case 4:
            printf("Thursday\n");
            break;
        case 5:
```

```
        printf("Friday\n");
        break;
      case 6:
        printf("Saturday\n");
        break;
      case 7:
        printf("Sunday\n");
        break;
      default:
        printf("Invalid day\n");
    }

    return 0;
}
```

Output: Wednesday

Q6:- Compare and contrast while loops, for loops, and do-while loops. Explain the scenarios in which each loop is most appropriate.

In C programming, loops are used to execute a block of code repeatedly based on a condition. The three primary types of loops are **while**, **for**, and **do-while**. Each loop has its own syntax and use cases, making them suitable for different scenarios. Below is a comparison of these loops, along with explanations of when to use each.

1. while Loop

Syntax:

```
while (condition) {
  // Code to execute while condition is true
}
```

Characteristics:

- The **while** loop checks the condition before executing the loop body.
- If the condition is false at the start, the loop body will not execute at all.

- It is suitable for situations where the number of iterations is not known in advance and depends on a condition.

Example:

```
#include <stdio.h>


int main() {
   int count = 1;


   while (count <= 5) {
      printf("%d\n", count);
      count++;
   }


   return 0;
}
```

Output:

```
Run code
1
2
3
4
5
```

When to Use:

- Use a **while** loop when the number of iterations is not predetermined and depends on a condition that may change during execution.

2. for Loop

Syntax:

```
for (initialization; condition; increment) {
   // Code to execute while condition is true
}
```

Characteristics:

- The **for** loop is typically used when the number of iterations is known beforehand.
- It combines initialization, condition checking, and increment/decrement in a single line, making it concise and easy to read.
- It is ideal for iterating over arrays or performing a specific number of iterations.

Example:

```
#include <stdio.h>


int main() {
    for (int i = 1; i <= 5; i++) {
        printf("%d\n", i);
    }


    return 0;
}
```

Output:

```
Run code
1
2
3
4
5
```

When to Use:

- Use a **for** loop when the number of iterations is known in advance, such as iterating through a fixed range or processing elements in an array.

3. do-while Loop

Syntax:

```
do {
    // Code to execute
} while (condition);
```

Characteristics:

- The **do-while** loop executes the loop body at least once before checking the condition.
- This guarantees that the code inside the loop will run at least once, regardless of whether the condition is true or false initially.
- It is useful when the loop body must be executed at least once, such as when prompting user input.

Example:

```
#include <stdio.h>

int main() {
    int count = 1;

    do {
        printf("%d\n", count);
        count++;
    } while (count <= 5);

    return 0;
}
```

Output:

Run code

```
1
2
3
4
5
```

When to Use:

- Use a **do-while** loop when you need to ensure that the loop body is executed at least once, such as when validating user input or performing an action that requires an initial execution.

Feature	**while Loop**	**for Loop**	**do-while Loop**
Condition Check	Before executing the loop body	Before executing the loop body	After executing the loop body
Execution Guarantee	May not execute if condition is false	Executes based on the condition	Always executes at least once
Syntax Convenience	More flexible, but less concise	Concise and structured	Simple, but less common
Use Case	Unknown iterations based on condition	Known iterations (fixed range)	At least one execution required

Q7:- Explain the use of break, continue, and goto statements in C. Provide examples of each.

In C programming, the **break**, **continue**, and **goto** statements are control flow statements that alter the normal flow of execution in loops and conditional structures. Here's a detailed explanation of each, along with examples.

1. break Statement

The **break** statement is used to exit from a loop or a switch statement prematurely. When a **break** statement is encountered, the control is transferred to the statement immediately following the loop or switch.

Example of break:

```
#include <stdio.h>


int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            break; // Exit the loop when i is 5
        }
        printf("%d\n", i);
    }
    return 0;
}
```

Output:

Run code

```
0
1
2
3
4
```

In this example, the loop prints numbers from 0 to 4. When **i** reaches 5, the **break** statement is executed, and the loop terminates.

2. continue Statement

The **continue** statement is used to skip the current iteration of a loop and proceed to the next iteration. When **continue** is encountered, the remaining code in the loop for that iteration is skipped.

Example of continue:

```
#include <stdio.h>

int main() {
    for (int i = 0; i < 10; i++) {
        if (i % 2 == 0) {
            continue; // Skip the even numbers
        }
        printf("%d\n", i);
    }
    return 0;
}
```

Output:

Run code

```
1

3

5

7

9
```

In this example, the loop prints only the odd numbers from 0 to 9. When **i** is even, the **continue** statement is executed, skipping the **printf** function for that iteration.

3. goto Statement

The **goto** statement is used to transfer control to a labeled statement in the program. It can lead to less readable code and is generally discouraged in favor of structured programming constructs like loops and functions.

Example of goto:

```
#include <stdio.h>


int main() {
   int i = 0;


   loop_start:
   if (i < 5) {
      printf("%d\n", i);
      i++;
      goto loop_start; // Jump back to the labeled statement
   }


   return 0;
}
```

Output:

Run code

```
0

1
```

```
2
3
4
```

In this example, the **goto** statement is used to create a loop that prints numbers from 0 to 4. The control jumps back to the **loop_start** label until the condition **i < 5** is false.

Q8 :- What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples.

In C programming, a function is a block of code that performs a specific task. Functions help in organizing code, making it reusable, and improving readability. A function typically consists of three main components: declaration, definition, and calling.

1. Function Declaration

A function declaration (also known as a function prototype) tells the compiler about the function's name, return type, and parameters (if any) without providing the actual body of the function. It is usually placed at the beginning of the program or in a header file.

Syntax:

```
return_type function_name(parameter_type1 parameter_name1, parameter_type2 parameter_name2, ...);
```

2. Function Definition

A function definition provides the actual body of the function, where the code to be executed is written. It includes the return type, function name, parameters, and the block of code that defines what the function does.

Syntax:

```
return_type function_name(parameter_type1 parameter_name1, parameter_type2 parameter_name2, ...) {
    // body of the function
    // return statement (if return_type is not void)
}
```

3. Calling a Function

To use a function, you need to call it from another function (usually from **main**). When calling a function, you provide the required arguments that match the parameters defined in the function.

Syntax:

```
function_name(argument1, argument2, ...);
```

Example:

```
#include <stdio.h>

// Function declaration
int add(int a, int b);

// Function definition
int add(int a, int b) {
   return a + b;
}

int main() {
   int result;

   // Calling the function
   result = add(5, 3);

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

Q9:- Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.

In C programming, an array is a collection of elements of the same data type, stored in contiguous memory locations. Arrays allow you to store multiple values in a single variable, making it easier to manage and manipulate data.

One-Dimensional Arrays

A one-dimensional array is the simplest form of an array, which can be thought of as a list of elements. It is defined by specifying the data type and the number of elements it can hold.

Declaration and Initialization:

```
#include <stdio.h>

int main() {

    // One-dimensional array

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

    printf("One-dimensional array:\n");

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

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

    }
```

Multi-Dimensional Arrays

Multi-dimensional arrays are arrays of arrays. The most common type is the two-dimensional array, which can be visualized as a table with rows and columns.

Declaration and Initialization:

```
#include <stdio.h>

int main() {

// Two-dimensional array

    int matrix[3][4] = {

        {1, 2, 3, 4},

        {5, 6, 7, 8},

        {9, 10, 11, 12}

    };

    printf("Two-dimensional array:\n");

    for (int i = 0; i < 3; i++) {

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

            printf("%d ", matrix[i][j]);

        }

        printf("\n");

    }

    return 0;

}
```

printf("%d", matrix[1][2]); // Outputs: 7

Q10:- Explain what pointers are in C and how they are declared and initialized. Why are pointers important in C?

Pointers in C are variables that store the memory address of another variable. They are a powerful feature of the C programming language, allowing for efficient manipulation of data and memory management.

Declaration of Pointers

To declare a pointer, you use the asterisk (*) symbol before the pointer's name. The syntax is as follows:

data_type *pointer_name;

For example, to declare a pointer to an integer, you would write:

int *ptr;

Initialization of Pointers

Pointers can be initialized by assigning them the address of a variable using the address-of operator (&).

For example:

int var = 10; // Declare an integer variable

int *ptr = &var; // Initialize pointer to the address of var

In this case, **ptr** now holds the memory address of **var**.

Importance of Pointers in C

1. **Dynamic Memory Management**: Pointers allow for dynamic memory allocation using functions like **malloc()**, **calloc()**, and **free()**. This is essential for creating data structures like linked lists, trees, and more.
2. **Efficient Array and String Manipulation**: Pointers can be used to iterate through arrays and strings efficiently without the need for indexing, which can be more performant in certain scenarios.
3. **Function Arguments**: Pointers enable passing large structures or arrays to functions without copying the entire data, which saves memory and processing time. This is often referred to as "pass by reference."
4. **Data Structures**: Pointers are fundamental in implementing complex data structures such as linked lists, trees, and graphs, where elements are dynamically linked.

5. **Low-Level Memory Access**: Pointers provide the ability to directly access and manipulate memory, which is crucial for systems programming and performance-critical applications.

Q11:- Explain string handling functions like strlen(), strcpy(), strcat(), strcmp(), and strchr(). Provide examples of when these functions are useful.

String handling functions are essential in C programming for manipulating and managing strings, which are essentially arrays of characters. Here's a brief overview of some commonly used string functions: **strlen()**, **strcpy()**, **strcat()**, **strcmp()**, and **strchr()**, along with examples of their usage.

1. strlen()

- **Description**: This function calculates the length of a string (number of characters before the null terminator).
- **Prototype**: **size_t strlen(const char *str);**
- **Use Case**: Useful when you need to determine how many characters are in a string, for example, to allocate memory dynamically.

Example:

```
#include <stdio.h>

#include <string.h>


int main() {

   const char *str = "Hello, World!";

   size_t length = strlen(str);

   printf("Length of the string: %zu\n", length); // Output: 13

   return 0;

}
```

2. strcpy()

- **Description**: This function copies a string from one location to another.
- **Prototype**: **char *strcpy(char *dest, const char *src);**
- **Use Case**: Useful for duplicating strings or initializing a string variable with another string.

Example:

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

```
#include <string.h>


int main() {
    char source[] = "Hello, World!";
    char destination[50];
    strcpy(destination, source);
    printf("Copied string: %s\n", destination); // Output: Hello, World!
    return 0;
}
```

3. strcat()

- **Description**: This function concatenates (appends) one string to the end of another.
- **Prototype**: **char *strcat(char *dest, const char *src);**
- **Use Case**: Useful for building strings dynamically, such as creating a full file path or combining user input.

Example:

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


int main() {
    char str1[50] = "Hello, ";
    char str2[] = "World!";
    strcat(str1, str2);
    printf("Concatenated string: %s\n", str1); // Output: Hello, World!
    return 0;
}
```

4. strcmp()

- **Description**: This function compares two strings lexicographically.
- **Prototype**: **int strcmp(const char *str1, const char *str2);**
- **Use Case**: Useful for sorting strings or checking if two strings are equal.

Example:

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

int main() {
    const char *str1 = "apple";
    const char *str2 = "banana";
    int result = strcmp(str1, str2);
    if (result < 0) {
        printf("%s is less than %s\n", str1, str2); // Output: apple is less than banana
    } else if (result > 0) {
        printf("%s is greater than %s\n", str1, str2);
    } else {
        printf("%s is equal to %s\n", str1, str2);
    }
    return 0;
}
```

5. strchr()

- **Description**: This function locates the first occurrence of a character in a string.
- **Prototype**: **char *strchr(const char *str, int c);**
- **Use Case**: Useful for parsing strings, such as finding delimiters in a CSV file.

Example:

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

int main() {
    const char *str = "Hello, World!";
    char *ptr = strchr(str, 'W');
    if (ptr != NULL) {
        printf("Found 'W' at position: %ld\n", ptr - str); // Output: Found 'W' at position:
7
```

```
    } else {
        printf("'W' not found\n");
    }
    return 0;
}
```

Q12:- Explain the concept of structures in C. Describe how to declare, initialize, and access structure members.

In C, a structure is a user-defined data type that allows you to group different types of variables under a single name. Structures are particularly useful for representing complex data types that consist of multiple attributes. Each variable within a structure is called a member.

Declaring a Structure

To declare a structure, you use the **struct** keyword followed by the structure name and the members enclosed in curly braces. Here’s the syntax:

```
struct StructureName {
    dataType member1;
    dataType member2;
    // ... other members
};
```

Example of Declaring a Structure

```
#include <stdio.h>


// Define a structure named 'Person'
struct Person {
    char name[50];
    int age;
    float height;
};
```

Initializing a Structure

You can initialize a structure at the time of declaration or later. Here are two ways to initialize a structure:

1. **At Declaration**: You can initialize the structure when you declare it.

   ```
   struct Person person1 = {"Alice", 30, 5.5};
   ```

2. **Using Designated Initializers**: You can also use designated initializers to specify which member to initialize.

   ```
   struct Person person2 = {.age = 25, .name = "Bob", .height = 6.0};
   ```

Accessing Structure Members

To access the members of a structure, you use the dot operator (**.**) if you have a structure variable. If you have a pointer to a structure, you use the arrow operator (**->**).

Example of Accessing Structure Members

```
#include <stdio.h>


int main() {
    struct Person person1 = {"Alice", 30, 5.5};


    // Accessing structure members
    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);
    printf("Height: %.2f\n", person1.height);


    // Modifying structure members
    person1.age = 31;
    printf("Updated Age: %d\n", person1.age);


    return 0;
}
```

Q13:- Explain the importance of file handling in C. Discuss how to perform file operations like opening, closing, reading, and writing files.

File handling in C is crucial for several reasons, as it allows programs to interact with external data stored in files. This capability is essential for tasks such as data persistence, configuration management, logging, and more. By using file handling, programs can read from and write to files, enabling them to store information beyond the program's execution time.

Importance of File Handling

1. **Data Persistence:** File handling allows data to be saved and retrieved even after the program has terminated. This is essential for applications that require data storage, such as databases and configuration files.
2. **Data Sharing:** Files enable data sharing between different programs or users. Multiple programs can read from or write to the same file, facilitating collaboration and data exchange.
3. **Logging:** Many applications use files to log events, errors, or user activities. This is important for debugging and monitoring application performance.
4. **Configuration Management:** Applications often read configuration settings from files, allowing users to customize behavior without modifying the source code.

File Operations in C

In C, file operations are performed using the standard library functions defined in the **<stdio.h>** header. Here are the basic operations:

1. Opening a File

To open a file, you use the **fopen()** function, which takes the filename and the mode as arguments. The mode specifies how the file will be used (e.g., read, write, append).

```
FILE *filePointer;

filePointer = fopen("example.txt", "r"); // Open for reading
```

Common modes include:

- **"r"**: Read mode (file must exist)
- **"w"**: Write mode (creates a new file or truncates an existing file)
- **"a"**: Append mode (writes data at the end of the file)
- **"r+"**: Read and write mode (file must exist)
- **"w+"**: Read and write mode (creates a new file or truncates an existing file)

2. Closing a File

After finishing file operations, it is important to close the file using the **fclose()** function to free resources.

```
fclose(filePointer);
```

3. Reading from a File

You can read data from a file using functions like **fgetc()**, **fgets()**, or **fread()**. Here’s an example using **fgets()** to read a line from a file:

```
char buffer[100];

if (fgets(buffer, sizeof(buffer), filePointer) != NULL) {

   printf("Read line: %s", buffer);

}
```

4. Writing to a File

To write data to a file, you can use functions like **fputc()**, **fputs()**, or **fwrite()**. Here’s an example using **fputs()** to write a string to a file:

```
FILE *filePointer;

filePointer = fopen("example.txt", "w"); // Open for writing

if (filePointer != NULL) {

   fputs("Hello, World!\n", filePointer);

   fclose(filePointer);

}
```