Tops Assignment: Basics of Computer Programming.

Module: 2 – Introduction to Programming

Q.1. **Write an essay covering the history and evolution of C programming. Explain its importance and why it is still used today.**

Ans.: The History and Evolution of C Programming

C began in the early 1970s when Dennis Ritchie at Bell Labs set out to create a language that could work well for system-level programming. At that time, operating systems were usually written in assembly language, which made them fast but difficult to maintain and less portable. Ritchie built C as an improvement over an earlier language called B, adding features like data types and better control structures. His goal was to design a language that was powerful but simple enough to write an entire operating system.

That idea turned out to be a turning point in computer science. In 1973, the UNIX operating system was rewritten in C, proving that a high-level language could be used to build something as large and complex as an OS. This established C as a practical and efficient systems language. Over the next decade, C spread rapidly through universities and industries. Brian Kernighan and Dennis Ritchie published *The C Programming Language* in 1978, often called "K&R C," which became the standard guide for learning the language.

As C grew more popular, a need for standardization became clear. Different compilers had slight variations in how they handled the language, which created portability issues. In 1989, ANSI (American National Standards Institute) introduced the first official standard known as ANSI C or C89. Later revisions followed, including C99 and C11, each adding features like improved libraries, inline functions, and better support for multi-threading. These updates kept C relevant while maintaining its core principles of efficiency and control.

Why C Is Important

C changed how software was built. It gave programmers the ability to work close to the hardware while still using a high-level language. Its design focuses on speed, small memory usage, and fine-grained control. These qualities made C ideal for writing operating systems, embedded systems, database engines, and compilers.

C also influenced many other languages. C++, Java, JavaScript, C#, and even Python borrow ideas from C, especially its syntax. Anyone who learns C finds it easier to understand how computers work internally, including memory management, pointers, and low-level operations.

Why C Is Still Used Today

Even after fifty years, C remains widely used. Several reasons explain its long life:

1. Speed and Performance:
   C programs run fast and use minimal memory, making the language perfect for systems

that must be reliable and efficient. This includes operating systems, device drivers, and real-time applications.

2. Hardware Control:
   C gives direct access to memory and hardware, something many modern languages hide. This makes C the preferred choice for embedded systems, IoT devices, and microcontroller programming.

3. Portability:
   With proper coding practices, a C program can run on almost any device. This level of portability is rare and extremely valuable.

4. Foundation for Other Languages:
   Compilers and interpreters for many modern languages are themselves written in C. Understanding C helps programmers learn how these tools work behind the scenes.

5. Large and Active Ecosystem:
   C has decades of libraries, tools, and community support. Its stability makes it a dependable option for long-term projects.

Conclusion

C programming has shaped the direction of modern computing. From its beginnings at Bell Labs to its role in building operating systems and embedded devices, C has proven to be powerful, flexible, and lasting. Its focus on performance and control keeps it essential even today. Whether someone wants to understand how computers work at a deeper level or build efficient software for real-world systems, C remains a language worth learning.

LAB EXERCISE:

Research and provide three real-world applications where C programming is extensively used, such as in embedded systems, operating systems, or game development.

Ans.: **1. Embedded Systems**

C is the backbone of embedded programming because it gives precise control over memory and hardware.
You'll find C in:

- Microcontrollers used in washing machines, microwaves, ACs, and automotive ECUs
- Medical devices like blood pressure monitors and ECG machines
- IoT devices like smart bulbs, smart meters, and wearables

Why C?
It runs fast, uses very little memory, and allows direct hardware interaction, which is essential in low-power, resource-limited devices.

---

2. Operating Systems

Most major operating systems have core components written in C.
Some examples:

- Linux kernel
- Windows core system components
- macOS and iOS kernel (XNU)
- Android's low-level system libraries

Why C?
It allows low-level control, efficient memory management, and portability across hardware platforms. OS developers need a language that can interact closely with hardware while staying fast and stable.

---

3. Game Development (Game Engines)

Many high-performance game engines and physics engines use C or C++.
Examples include:

- Doom engine
- Quake engine
- Godot's core modules
- Nintendo consoles often use C/C++ for game logic

Why C?
Games need high speed, optimized performance, and real-time rendering. C offers the ability to fine-tune performance at the hardware level, which other high-level languages can't match.

Q.2. 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.

Ans: Steps to Install a C Compiler (GCC) and Set Up an IDE

1. Installing GCC (Windows)

1. Download **MinGW-w64** from its official website.
2. Run the installer and choose:
   - Architecture: x86_64
   - Threads: posix
   - Exception: seh
3. Select an installation directory, usually C:\mingw-w64.
4. After installation, add the bin folder path (for example C:\mingw-w64\bin) to your **System Environment Variables** so you can use gcc from the command line.

5. Open Command Prompt and type:

```
gcc --version
```

If it shows a version, the installation is successful.

2. Setting Up DevC++

1. Download DevC++ and install it.
2. Open the IDE and create a new C project or source file.
3. DevC++ already includes a compiler, so no extra setup is required.
4. Write a C program and click **Compile & Run** to execute it.

---

3. Setting Up CodeBlocks

1. Download **CodeBlocks with MinGW** (very important because it includes the compiler).
2. Install it and choose the default settings.
3. Open CodeBlocks, go to **Settings > Compiler**, and make sure the GNU GCC compiler is detected.
4. Create a new C file, write your program, and click **Build and Run**.

---

4. Setting Up VS Code

1. Install **VS Code**.
2. Install the extension **C/C++** from Microsoft.
3. Install MinGW-w64 (as described earlier) if you haven't already.
4. In VS Code, create a folder and add a .c file.
5. Configure tasks.json and launch.json (VS Code will guide you automatically when you click Run).
6. Press **Run > Run Without Debugging** to execute your C file.

Q.3. Explain the basic structure of a C program, including headers, main function, comments, data types, and variables. Provide examples.

Ans: Basic Structure of a C Program

A C program follows a standard structure. It includes headers, a main function, variables, data types, and comments. Each part plays a specific role in how the program works.

---

1. Header Files

Header files tell the compiler what built-in functions you want to use.
They are included at the top of the program using #include.

Example:

```c
#include <stdio.h>  // allows use of printf(), scanf()
```

2. Comments

Comments help you explain your code.
They are ignored by the compiler.

Two types:

- Single-line comment:

```c
// This is a comment
```

- Multi-line comment:

```c
/* This is a
   multi-line comment */
```

3. main() Function

This is the starting point of every C program.
The compiler begins execution from the main function.

Example:

```c
int main() {
    // code goes here
    return 0;
}
```

return 0; means the program ended successfully.

4. Data Types

Data types tell the compiler what kind of data a variable will hold.

Common data types:

- int → whole numbers (10, -2, 400)
- float → decimal numbers (3.14, 9.8)
- char → single character ('A', 'b', '3')
- double → large decimal numbers (precision higher than float)

---

5. Variables

A variable is a container that stores a value.
Every variable must have:

- a data type
- a name
- a value (optional during declaration)

Example:

```c
int age = 25;
float price = 99.50;
char grade = 'A';
```

Complete Example of a C Program:

```c
#include <stdio.h>  // header file

// This program prints user information
int main() {
    int age = 22;          // variable of type int
    float height = 5.8;    // variable of type float
    char grade = 'A';      // variable of type char

    printf("Age: %d\n", age);
    printf("Height: %.1f\n", height);
    printf("Grade: %c\n", grade);

    return 0;
}
```

Explanation of the Example:

- #include <stdio.h> allows you to use printf().
- Comments explain what the program does.

- Variables store data.
- The main() function runs everything.

Q.4. Write notes explaining each type of operator in C: arithmetic, relational, logical, assignment, increment/decrement, bitwise, and conditional operators

Ans.: Operators in C – Notes

Operators are symbols that tell the compiler to perform specific actions on variables or values. C provides different categories of operators depending on the type of operation.

1. Arithmetic Operators

These operators perform basic mathematical operations.

Operator	Meaning	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (remainder)	a % b

2. Relational Operators

These compare two values and return either true (1) or false (0).

Operator	Meaning	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b

3. Logical Operators

Used to combine or check multiple conditions.

Operator	Meaning	Example
&&	Logical AND	(a > 0 && b > 0)
||	Logical OR	` (a > 0
!	Logical NOT	!(a > b)

4. Assignment Operators

These assign values to variables.

Operator	Meaning	Example
=	Assign	a = 10
+=	Add and assign	a += 5 (same as a = a + 5)
-=	Subtract and assign	a -= 3
*=	Multiply and assign	a *= 2
/=	Divide and assign	a /= 2
%=	Modulus and assign	a %= 3

5. Increment and Decrement Operators

Used to increase or decrease a variable by 1.

Operator	Meaning
++	Increment
--	Decrement

Two types:

- **Pre-increment:** ++a (increase first, then use)
- **Post-increment:** a++ (use first, then increase)

Example:

6. Bitwise Operators

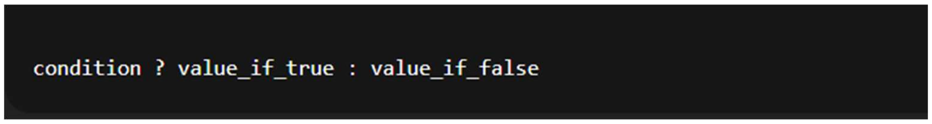
These work on the binary (bit level) representation of numbers.

Operator	Meaning	Example
&	Bitwise AND	a & b
|	Bitwise OR	` a
^	Bitwise XOR	a ^ b
<<	Left shift	a << 1
>>	Right shift	a >> 1
~	Bitwise NOT	~a

7. Conditional (Ternary) Operator

This is a compact form of an if-else statement.

Syntax:



```
condition ? value_if_true : value_if_false
```

LAB EXERCISE:

Write a C program that accepts two integers from the user and performs arithmetic, relational, and logical operations on them. Display the results.

Ans:

```
#include <stdio.h>

int main ()

{

 int a, b;

 printf("enter values of a and b \n");

 scanf("%d %d", &a, &b);

 // sum=a+b;

 printf("\nsum=%d", a + b);
```

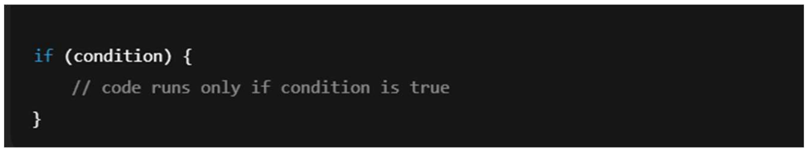
```
  printf("\nsub=%d", a - b);

  printf("\nmul=%d", a * b);

  printf("\ndiv=%f", (float)a / b);

  printf("\nmodulo=%d", a % b);

  return 0;

}


/*

output:

---------

enter values of a and b

34

43


sum=77

sub=-9

mul=1462

div=0.790698

modulo=34

*/
```

Q.5. Explain decision-making statements in C (if, else, nested if-else, switch). Provide examples of each.

Ans. 1. if Statement

Used when you want to execute some code only if a condition is true.

Syntax :



```
if (condition) {
    // code runs only if condition is true
}
```

Example:

```c
#include <stdio.h>

int main() {
    int age = 20;

    if (age >= 18) {
        printf("You can vote.\n");
    }

    return 0;
}
```

2. if-else Statement

Used when you have two choices:
– run something if the condition is true
– run something else if the condition is false

Syntax:

```c
if (condition) {
    // true case
} else {
    // false case
}
```

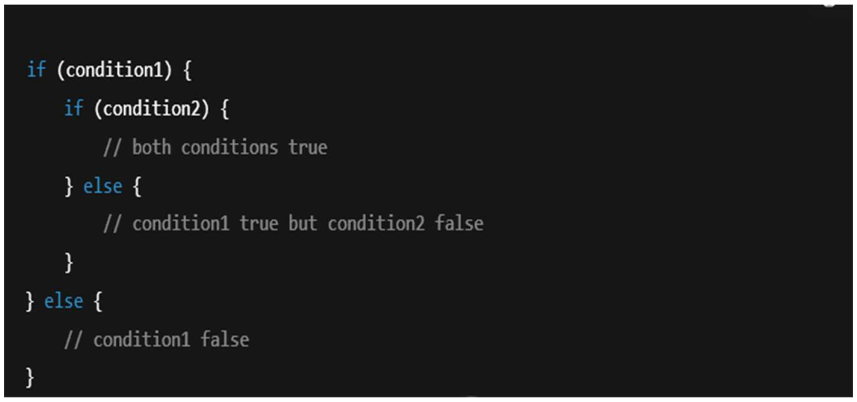
Example:

```c
#include <stdio.h>

int main() {
    int marks = 35;

    if (marks >= 40) {
        printf("Pass\n");
    } else {
        printf("Fail\n");
    }

    return 0;
}
```

3. Nested if-else

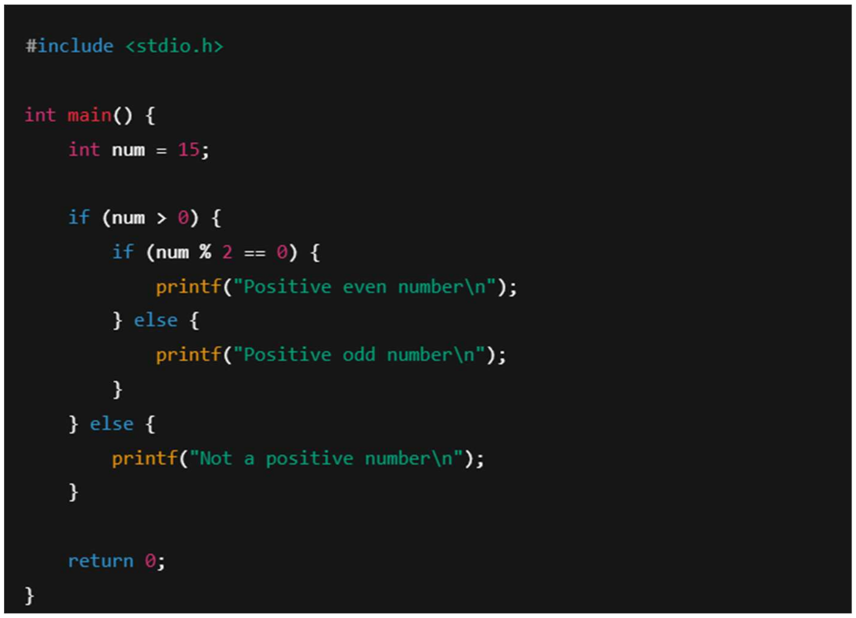
An if inside another if.
Used when you need multiple conditions checked step by step.

Syntax :



```c
if (condition1) {
    if (condition2) {
        // both conditions true
    } else {
        // condition1 true but condition2 false
    }
} else {
    // condition1 false
}
```

Example:

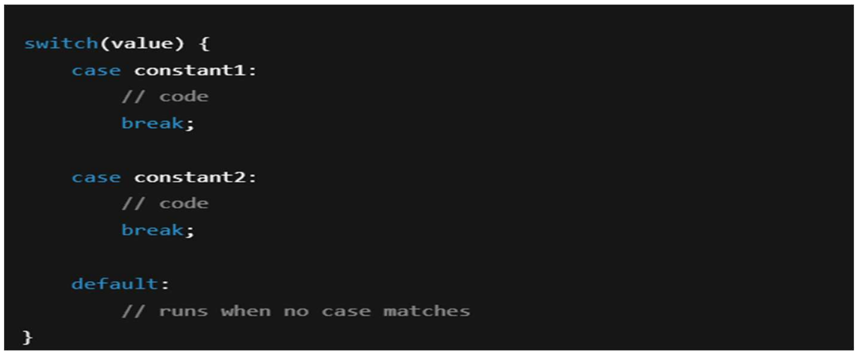


```c
#include <stdio.h>

int main() {
    int num = 15;

    if (num > 0) {
        if (num % 2 == 0) {
            printf("Positive even number\n");
        } else {
            printf("Positive odd number\n");
        }
    } else {
        printf("Not a positive number\n");
    }

    return 0;
}
```

4. switch Statement

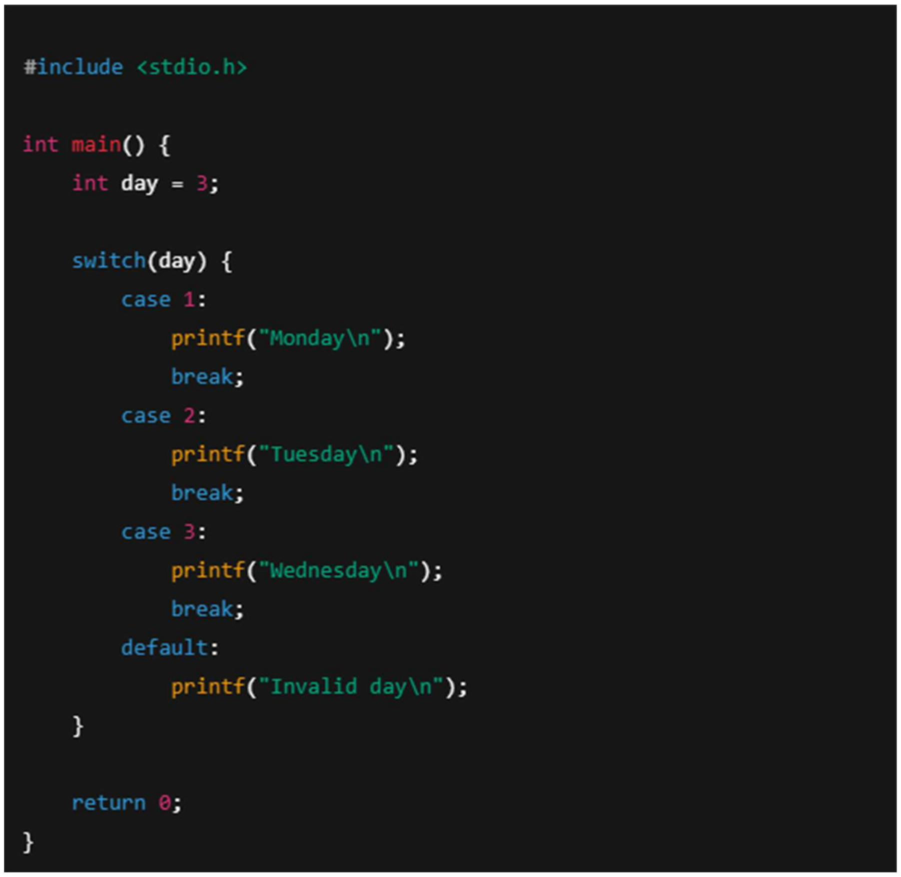
Used when you need to compare one value with many possible cases.
It’s cleaner than writing many if-else-if.

Syntax:



```c
switch(value) {
    case constant1:
        // code
        break;

    case constant2:
        // code
        break;

    default:
        // runs when no case matches
}
```

Example:

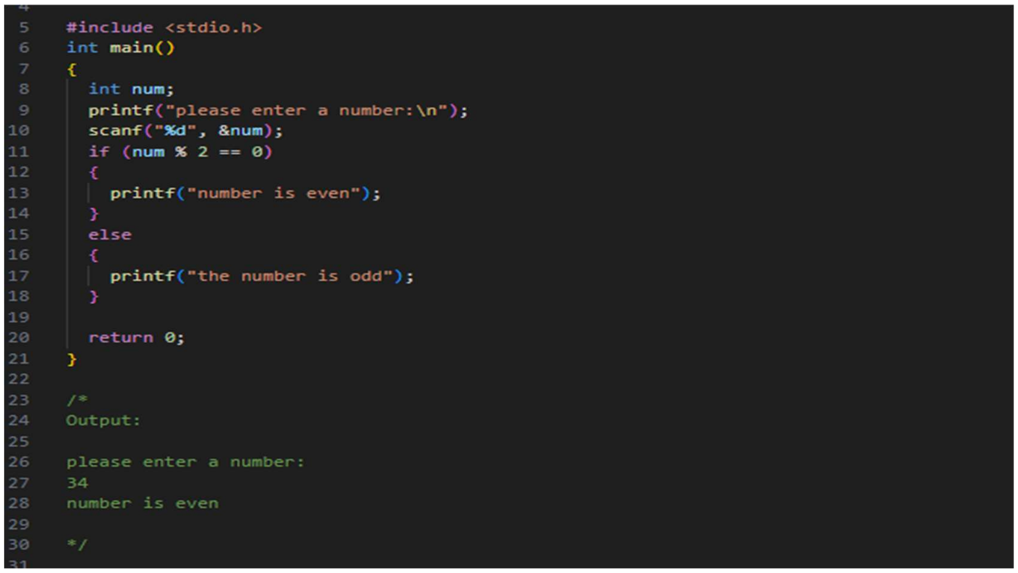


```c
#include <stdio.h>

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;
        default:
            printf("Invalid day\n");
    }

    return 0;
}
```

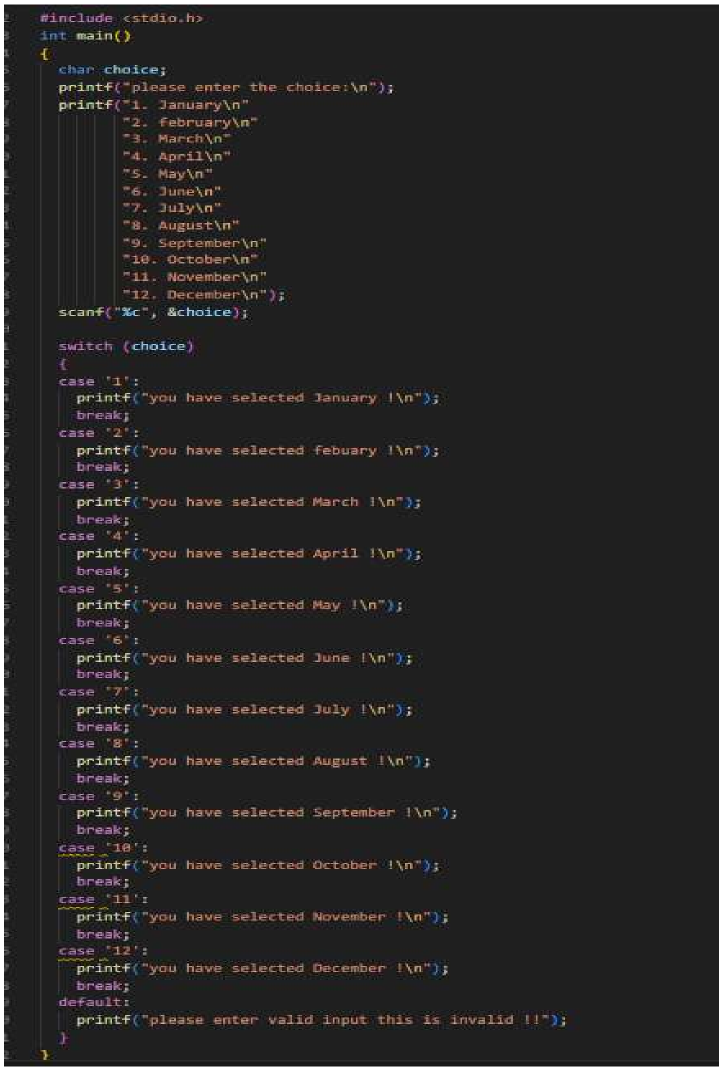
LAB EXERCISES:

Write a C program to check if a number is even or odd using an if-else statement. Extend the program using a switch statement to display the month name based on the user's input (1 for January 2 for February, etc.).

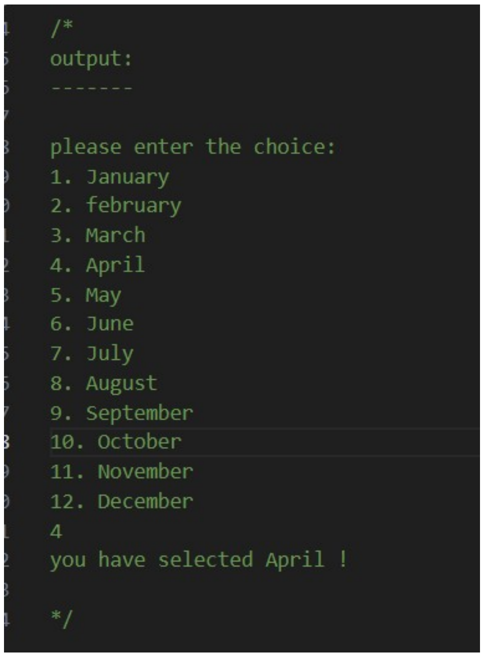
Ans:



```c
#include <stdio.h>
int main()
{
  int num;
  printf("please enter a number:\n");
  scanf("%d", &num);
  if (num % 2 == 0)
  {
    printf("number is even");
  }
  else
  {
    printf("the number is odd");
  }

  return 0;
}

/*
Output:

please enter a number:
34
number is even

*/
```



```c
#include <stdio.h>
int main()
{
  char choice;
  printf("please enter the choice:\n");
  printf("1. January\n"
           "2. february\n"
           "3. March\n"
           "4. April\n"
           "5. May\n"
           "6. June\n"
           "7. July\n"
           "8. August\n"
           "9. September\n"
           "10. October\n"
           "11. November\n"
           "12. December\n");
  scanf("%c", &choice);

  switch (choice)
  {
  case '1':
    printf("you have selected January !\n");
    break;
  case '2':
    printf("you have selected febuary !\n");
    break;
  case '3':
    printf("you have selected March !\n");
    break;
  case '4':
    printf("you have selected April !\n");
    break;
  case '5':
    printf("you have selected May !\n");
    break;
  case '6':
    printf("you have selected June !\n");
    break;
  case '7':
    printf("you have selected July !\n");
    break;
  case '8':
    printf("you have selected August !\n");
    break;
  case '9':
    printf("you have selected September !\n");
    break;
  case '10':
    printf("you have selected October !\n");
    break;
  case '11':
    printf("you have selected November !\n");
    break;
  case '12':
    printf("you have selected December !\n");
    break;
  default:
    printf("please enter valid input this is invalid !!");
  }
}
```



```
/*
output:
-------

please enter the choice:
1. January
2. february
3. March
4. April
5. May
6. June
7. July
8. August
9. September
10. October
11. November
12. December
4
you have selected April !

*/
```

Q.6. Compare and contrast while loops, for loops, and do-while loops. Explain the scenarios in which each loop is most appropriate.

Ans: **1. While Loop**

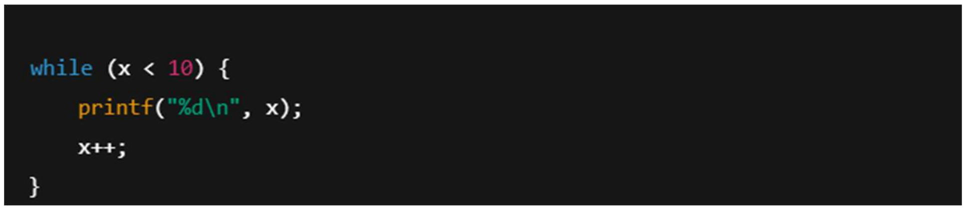
How it works

- Checks the condition first.
- If the condition is true, the loop runs.
- If the condition is false from the beginning, the loop never runs.

When to use

- When you **don't know beforehand** how many times the loop should run.
- When the loop depends on a condition that changes inside the loop (like user input or sensor value).
- Good for waiting-type logic.

Example:



```
while (x < 10) {
    printf("%d\n", x);
    x++;
}
```

2. For Loop

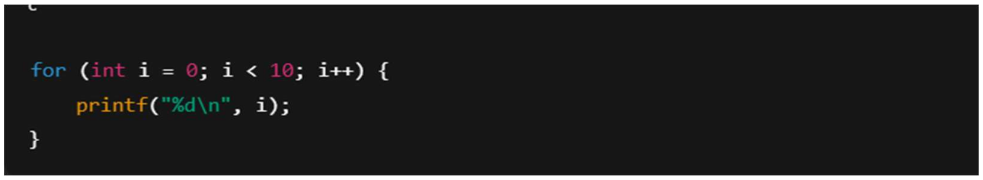
How it works

- Has initialization, condition, and update in a single line.
- Condition is checked before each iteration.
- Mostly used for count-controlled loops.

When to use

- When you know the exact number of iterations.
- When you're working with counters, arrays, ranges, tables, patterns.
- It keeps the loop control variables neatly organized.

Example:



```
for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

3. Do-While Loop

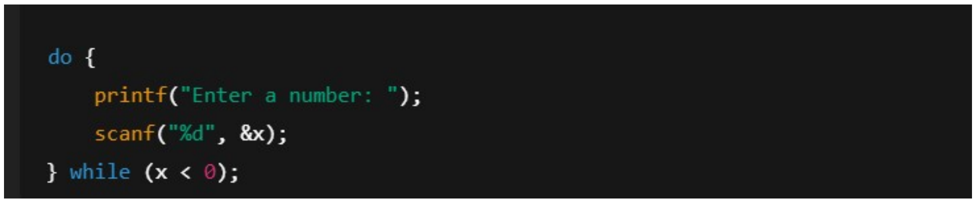
How it works

- Executes the loop body once before checking the condition.
- After the first run, it repeats only if the condition is true.

When to use

- When you want the code to run at least once, no matter what.
- Ideal for menus, repeat-until tasks, getting user confirmation.

Example:



```
do {
    printf("Enter a number: ");
    scanf("%d", &x);
} while (x < 0);
```

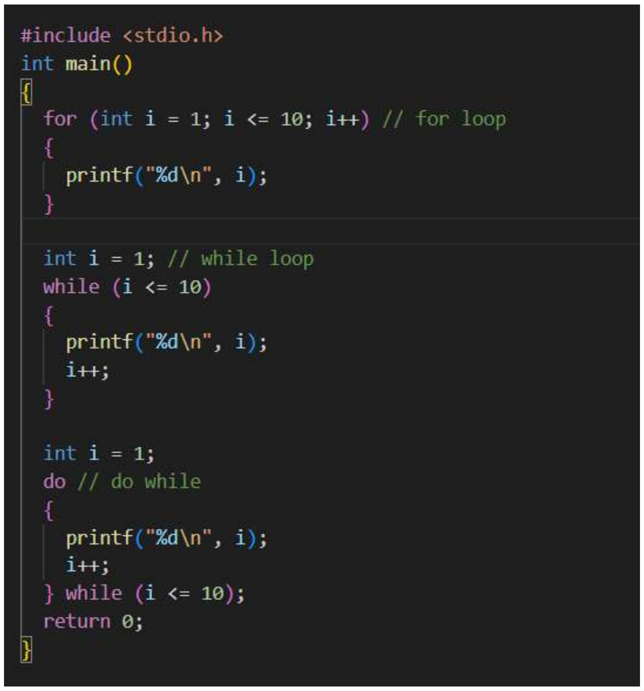
Quick Comparison Table:

Feature	while	for	do-while
Condition checked	Before	Before	After
Runs at least once	No	No	Yes
Best use case	Unknown repetition count	Known repetition count	Must run once (menus, inputs)
Readability	Good for simple conditions	Best for counters	Good for user-driven loops

LAB EXERCISE:

Write a C program to print numbers from 1 to 10 using all three types of loops (while, for, do-while).

Ans:



```c
#include <stdio.h>
int main()
{
  for (int i = 1; i <= 10; i++) // for loop
  {
    printf("%d\n", i);
  }

  int i = 1; // while loop
  while (i <= 10)
  {
    printf("%d\n", i);
    i++;
  }

  int i = 1;
  do // do while
  {
    printf("%d\n", i);
    i++;
  } while (i <= 10);
  return 0;
}
```

Output: *this output is getting after one by one run the each loops type .*

```
1

2

3

4

5

6

7

8

9

10
```

Q.7. Explain the use of break, continue, and goto statements in C. Provide examples of each.

Ans:

1. break

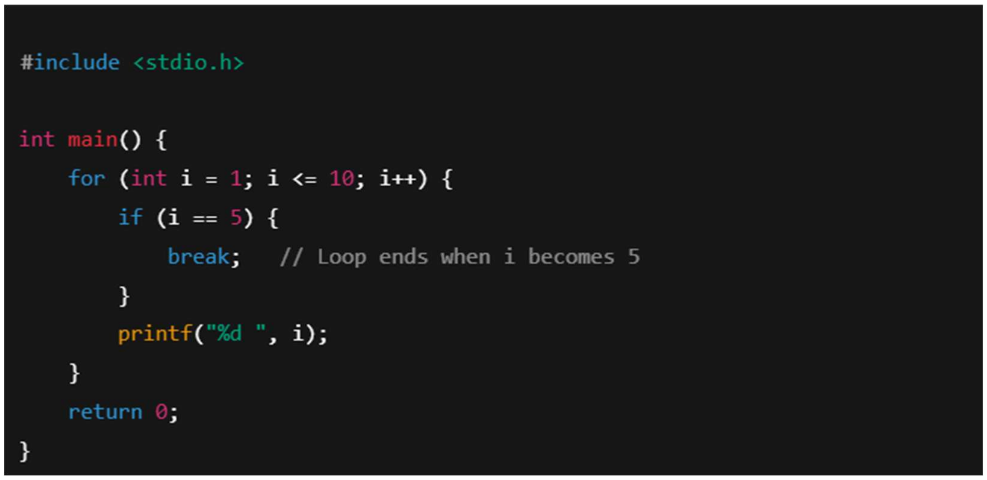
What it does

It immediately stops the nearest loop (for, while, do-while) or switch and exits out of it.

When to use

- When you want to stop a loop early based on a condition.
- When you want to exit a switch-case.

Example: Using break in a loop



```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            break;   // Loop ends when i becomes 5
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output:
1 2 3 4
(The loop stops before printing 5)

2. continue

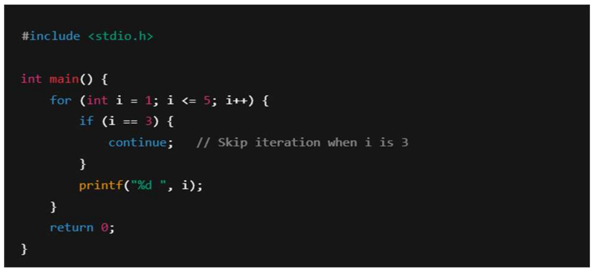
What it does

It skips the current iteration of the loop and moves directly to the next iteration.

When to use

- When you want to ignore one particular iteration.
- Useful when you want to skip some values.

Example: Using continue



```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue;   // Skip iteration when i is 3
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output:
1 2 4 5
(3 is skipped)

3. goto

What it does

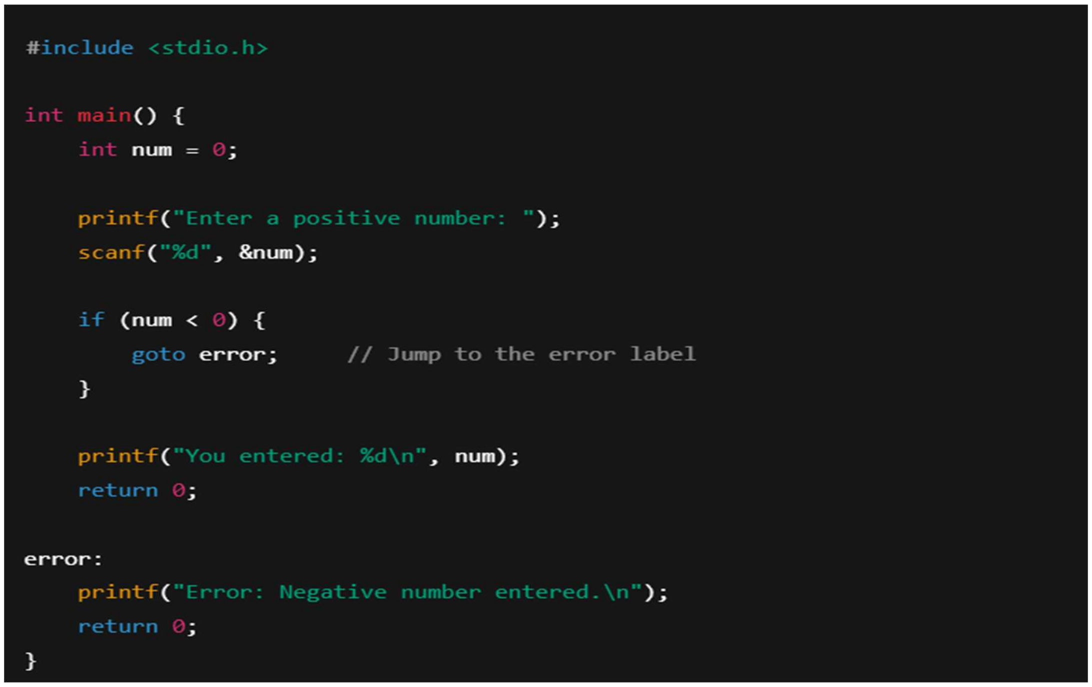
It jumps to a labeled statement anywhere in the same function.

When to use

- For breaking out of multiple nested loops at once.
- For error handling in large programs.

But generally, avoid it unless needed. It can make the code harder to read.

Example: Using goto



```c
#include <stdio.h>

int main() {
    int num = 0;

    printf("Enter a positive number: ");
    scanf("%d", &num);

    if (num < 0) {
        goto error;      // Jump to the error label
    }

    printf("You entered: %d\n", num);
    return 0;

error:
    printf("Error: Negative number entered.\n");
    return 0;
}
```

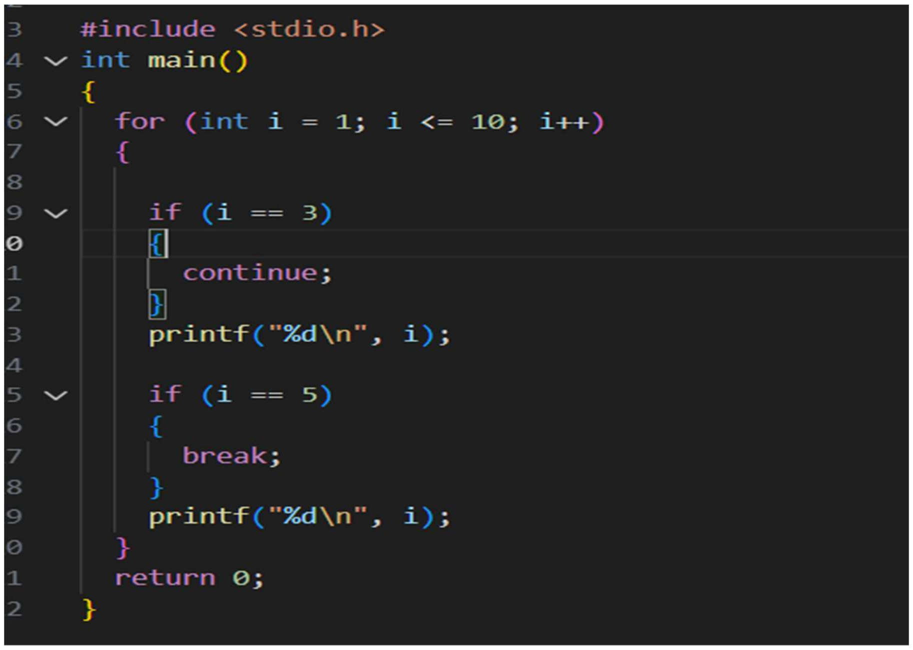
Statement	**Purpose**	**Works In**	**Effect**
break	Stop loop/switch	loops, switch	Exits loop completely
continue	Skip iteration	loops	Skips to next iteration

Statement	Purpose	Works In	Effect
goto	Jump to label	same function	Jumps anywhere (use rarely)

LAB EXERCISE:

Write a C program that uses the break statement to stop printing numbers when it reaches 5. Modify the program to skip printing the number 3 using the continue statement.

Ans:



```c
#include <stdio.h>
int main()
{
  for (int i = 1; i <= 10; i++)
  {

    if (i == 3)
    {
      continue;
    }
    printf("%d\n", i);

    if (i == 5)
    {
      break;
    }
    printf("%d\n", i);
  }
  return 0;
}
```

Q.8. What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples.

Ans: 1. Function Components in C

A) Function Declaration (Prototype)

This tells the compiler:

- the function name
- return type

- parameters (if any)

It is usually written before main() or in a header file.

Syntax:

```
return_type function_name(parameter_list);
```

Example:

```
int add(int a, int b);
```

B) Function Definition

This is the actual body of the function.
It contains the code that runs when the function is called.

Syntax:

```
return_type function_name(parameter_list) {
    // code
}
```

Example:

```
int add(int a, int b) {
    return a + b;
}
```

C) Calling a Function

This is how you execute the function inside main() or another function.

Syntax:

```
function_name(arguments);
```

Example:

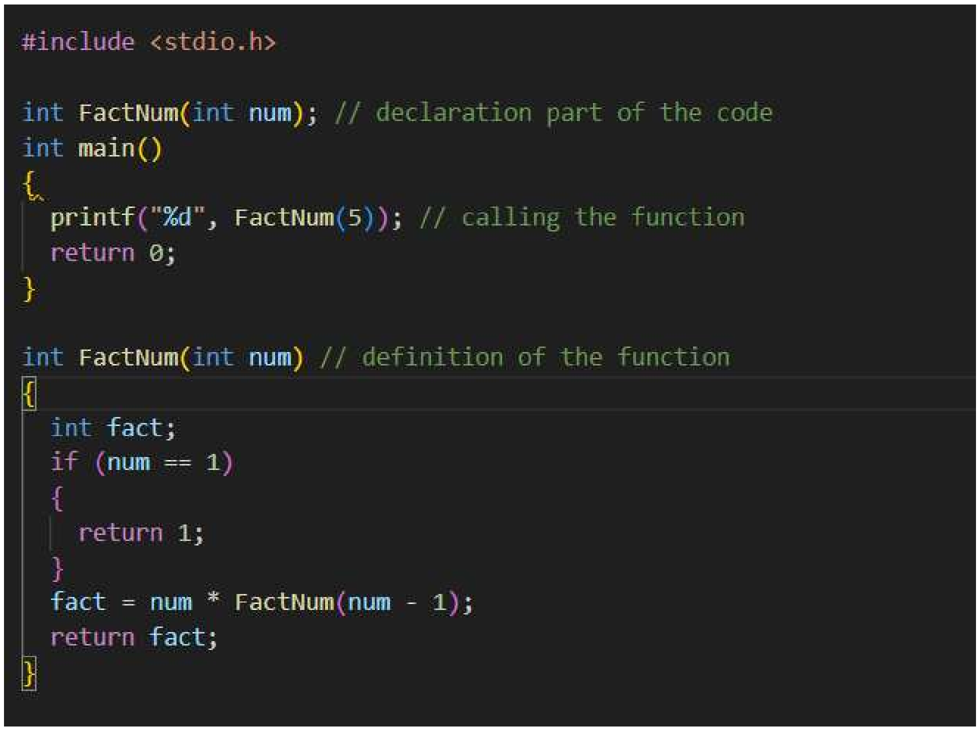
```
int result = add(5, 3);
```

Statement	Purpose	Works In	Effect
break	Stop loop/switch	loops, switch	Exits loop completely
continue	Skip iteration	loops	Skips to next iteration
goto	Jump to label	same function	Jumps anywhere (use rarely)

LAB EXERCISE:

Write a C program that calculates the factorial of a number using a function. Include function declaration, definition, and call.

Ans:



```c
#include <stdio.h>

int FactNum(int num); // declaration part of the code
int main()
{
  printf("%d", FactNum(5)); // calling the function
  return 0;
}

int FactNum(int num) // definition of the function
{
  int fact;
  if (num == 1)
  {
    return 1;
  }
  fact = num * FactNum(num - 1);
  return fact;
}
```

Output :

```
120
```

Q.9. Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.

Ans: Arrays:

An **array** in C is a collection of elements of the **same data type** stored in **contiguous memory locations**.
Each element is accessed using an **index**, and indexing always starts from **0**.

Arrays are used when you want to store and process multiple values efficiently using a single variable name.

Why use arrays?

- Store multiple values of the same type
- Reduce code repetition
- Easy traversal using loops
- Efficient memory usage

General Syntax:

```
data_type array_name[size];
```

Examples:

```
int marks[5];
```

1. One-Dimensional Array (1D Array) :

A one-dimensional array is like a **list** or **linear sequence** of elements.

Use case:
Marks of students, list of prices, scores, ages, etc.

Declaration:

```
int arr[5];
```

Initialization:

```
int arr[5] = {10, 20, 30, 40, 50};
```

Accessing elements:

```c
printf("%d", arr[2]);   // Output: 30
```

Example Program:

```c
#include <stdio.h>

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

    for(int i = 0; i < 5; i++) {
        printf("%d ", numbers[i]);
    }

    return 0;
}
```

2. Multi-Dimensional Array :

A multi-dimensional array contains **more than one dimension**.
The most common is the **two-dimensional array (2D array)**, which looks like a **table or matrix** (rows and columns).

Use case:
Tables, matrices, student marks (rows = students, columns = subjects), games grids, etc.

Declaration:

```c
int matrix[3][3];
```

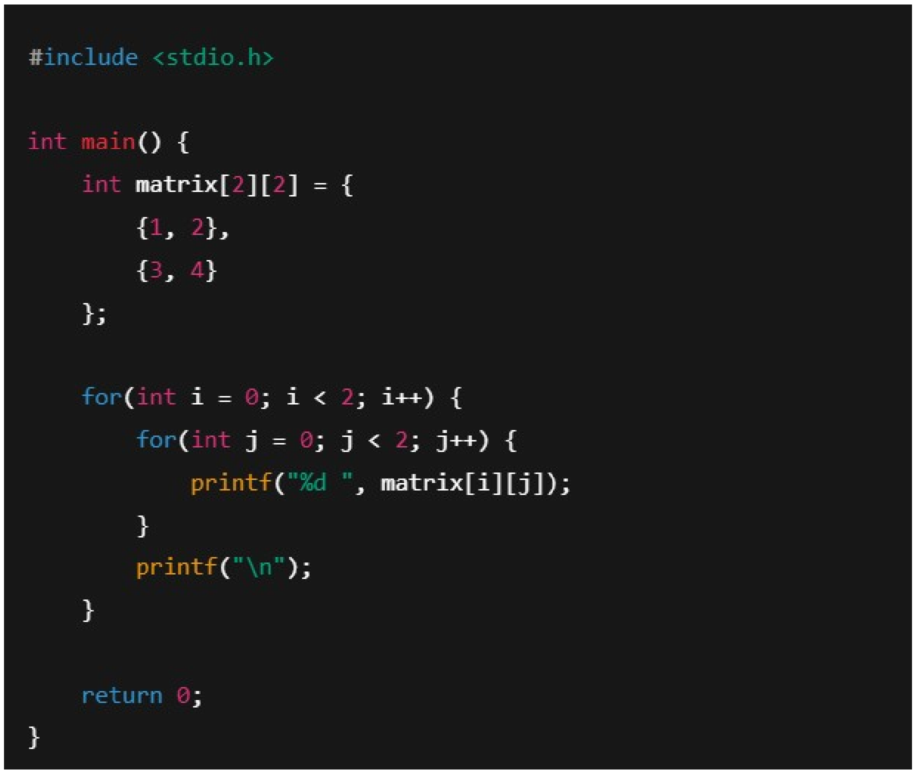
Initialization:

```c
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Accessing elements:

```c
printf("%d", matrix[1][2]);   // Output: 6
```

Example program:



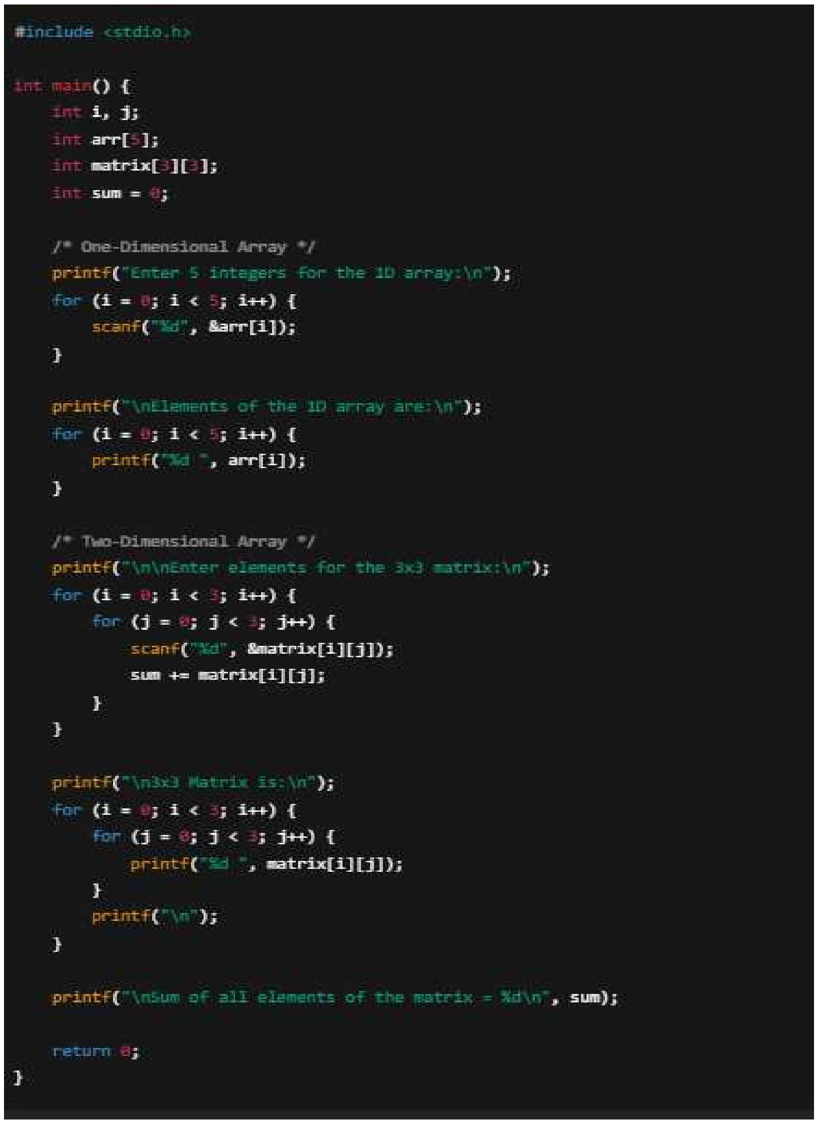
```c
#include <stdio.h>

int main() {
    int matrix[2][2] = {
        {1, 2},
        {3, 4}
    };

    for(int i = 0; i < 2; i++) {
        for(int j = 0; j < 2; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Difference Between One-Dimensional and Multi-Dimensional Arrays

Feature	**One-Dimensional Array**	**Multi-Dimensional Array**
Dimensions	Single	Two or more
Structure	Linear list	Table / Matrix
Indexing	One index	Two or more indices
Example	arr[5]	arr[3][4]
Loop required	Single loop	Nested loops
Memory layout	Continuous	Continuous (row-wise)

LAB EXERCISE:

Write a C program that stores 5 integers in a one-dimensional array and prints them. Extend this to handle a two-dimensional array (3x3 matrix) and calculate the sum of all elements.



```c
#include <stdio.h>

int main() {
    int i, j;
    int arr[5];
    int matrix[3][3];
    int sum = 0;

    /* One-Dimensional Array */
    printf("Enter 5 integers for the 1D array:\n");
    for (i = 0; i < 5; i++) {
        scanf("%d", &arr[i]);
    }

    printf("\nElements of the 1D array are:\n");
    for (i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    /* Two-Dimensional Array */
    printf("\n\nEnter elements for the 3x3 matrix:\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            scanf("%d", &matrix[i][j]);
            sum += matrix[i][j];
        }
    }

    printf("\n3x3 Matrix is:\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    printf("\nSum of all elements of the matrix = %d\n", sum);

    return 0;
}
```

```
/* output

Enter 5 integers for the 1D array:

1 2 3 4 5


Elements of the 1D array are:

1 2 3 4 5


Enter elements for the 3x3 matrix:

1 2 3
```

4 5 6

7 8 9

3x3 Matrix is:

1 2 3

4 5 6

7 8 9

Sum of all elements of the matrix = 45

*/

Q.10. Explain what pointers are in C and how they are declared and initialized. Why are pointers important in C?

Ans: A **pointer** is a variable that **stores the memory address of another variable** instead of storing a value directly.

A pointer *points* to the location where data is stored in memory.

Declaration of a Pointer:

Syntax:

```
data_type *pointer_name;
```

Example:

```
int *p;
```

Here:

- int → type of data the pointer will point to
- *p → pointer variable
- p can store the address of an integer variable

Initialization of a Pointer

A pointer is initialized by assigning it the **address of a variable** using the **address-of operator (&)**.

Example:

```c
int a = 10;
int *p;

p = &a;
```

Now:

- p stores the address of a
- *p gives the value stored at that address (10)

Using a Pointer (Dereferencing)

The **dereference operator (*)** is used to access the value at the address stored in the pointer.

```c
printf("%d", *p);   // Output: 10
```

Example Program:

```c
#include <stdio.h>

int main() {
    int a = 25;
    int *p;

    p = &a;

    printf("Value of a = %d\n", a);
    printf("Address of a = %p\n", &a);
    printf("Value stored in pointer p = %p\n", p);
    printf("Value pointed by p = %d\n", *p);

    return 0;
}
```

Why Are Pointers Important in C?

Pointers are one of the **most powerful features of C**. They are important because:

1. Efficient Memory Management

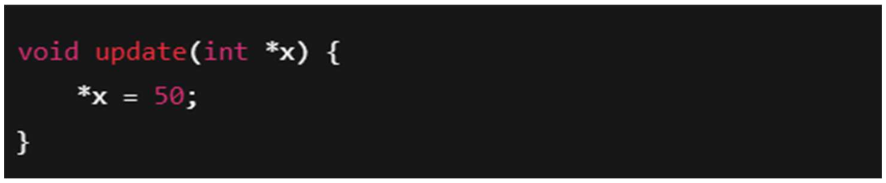
- Pointers allow **direct access to memory**
- Used in dynamic memory allocation (malloc, calloc, free)

2. Function Call by Reference

- Allows functions to **modify actual variables**

- Saves memory by avoiding data copying

Example:



```
void update(int *x) {
    *x = 50;
}
```

3. Arrays and Strings

- Arrays are accessed using pointers
- Strings are handled using character pointers

4. Dynamic Data Structures

- Essential for:
    - Linked lists
    - Stacks
    - Queues
    - Trees
    - Graphs

5. Faster Program Execution

- Passing pointers to functions is faster than passing large variables

LAB EXERCISE:

Write a C program to demonstrate pointer usage. Use a pointer to modify the value of a variable and print the result.

```c
#include <stdio.h>

int main() {
    int num = 10;
    int *ptr;

    /* Pointer initialization */
    ptr = &num;

    /* Modifying value using pointer */
    *ptr = 25;

    /* Output */
    printf("Value of num after modification = %d\n", num);

    return 0;
}
```

Output:

Value of num after modification = 25

Q.11. Explain string handling functions like strlen(), strcpy(), strcat(), strcmp(), and strchr(). Provide examples of when these functions are useful.

Ans: Strings are arrays of characters terminated by a null character '\0'.
The **string handling functions** are provided by the header file:

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

1. strlen() – String Length Function:

- strlen() returns the length of a string excluding the null character '\0'.

Use

- To find length of input string
- To validate password or username length
- To control loops on strings

2. strcpy() – String Copy Function :

- strcpy() copies one string into another string.

```c
strcpy(destination, source);
```

Use

- Copying user input
- Assigning string values
- Duplicating strings

Destination array must be large enough.

3. strcat() – String Concatenation Function :

- strcat() appends one string at the end of another string.

```
strcat(destination, source);
```

Use

- Joining first name and last name
- Creating messages
- Combining strings

4. strcmp() – String Compare Function :

- strcmp() compares two strings lexicographically.

```
strcmp(string1, string2);
```

Return Values

- 0 → Strings are equal
- < 0 → First string is smaller
- > 0 → First string is greater

Use

- Login authentication
- Comparing commands
- Sorting strings

5. strchr() – Character Search Function:

- strchr() finds the first occurrence of a character in a string.

```
strchr(string, character);
```

Use

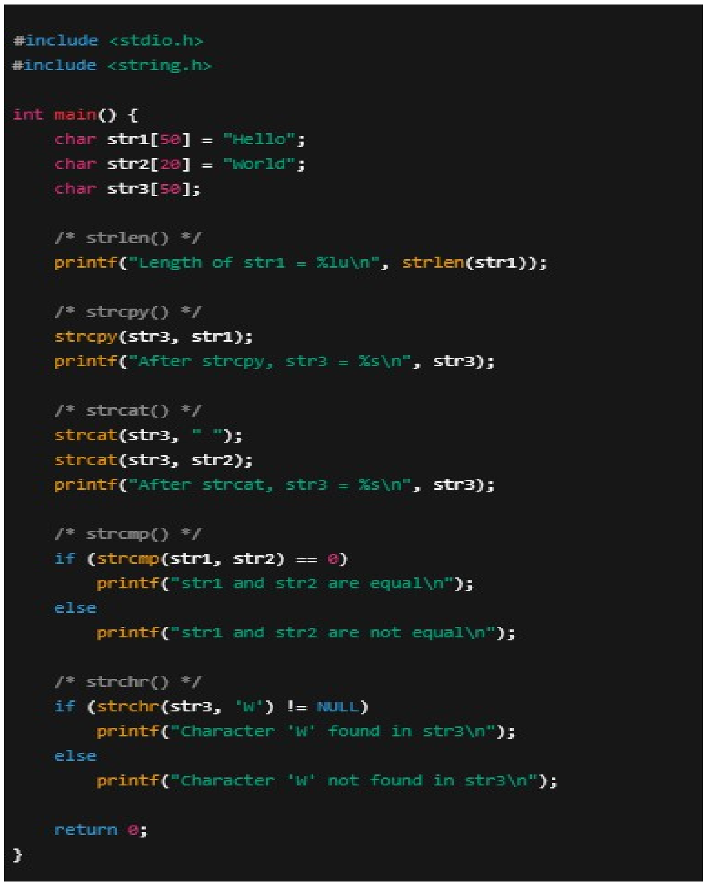
- **Searching specific characters**
- **Parsing strings**
- **Input validation**

Comparison table :

Function	Purpose	Header File
strlen()	Find string length	<string.h>
strcpy()	Copy string	<string.h>
strcat()	Join strings	<string.h>
strcmp()	Compare strings	<string.h>
strchr()	Find character in string	<string.h>

Examples of all in one:

Code:



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

int main() {
    char str1[50] = "Hello";
    char str2[20] = "World";
    char str3[50];

    /* strlen() */
    printf("Length of str1 = %lu\n", strlen(str1));

    /* strcpy() */
    strcpy(str3, str1);
    printf("After strcpy, str3 = %s\n", str3);

    /* strcat() */
    strcat(str3, " ");
    strcat(str3, str2);
    printf("After strcat, str3 = %s\n", str3);

    /* strcmp() */
    if (strcmp(str1, str2) == 0)
        printf("str1 and str2 are equal\n");
    else
        printf("str1 and str2 are not equal\n");

    /* strchr() */
    if (strchr(str3, 'W') != NULL)
        printf("Character 'W' found in str3\n");
    else
        printf("Character 'W' not found in str3\n");

    return 0;
}
```

/* Outputs:

Length of str1 = 5

After strcpy, str3 = Hello

After strcat, str3 = Hello World

str1 and str2 are not equal

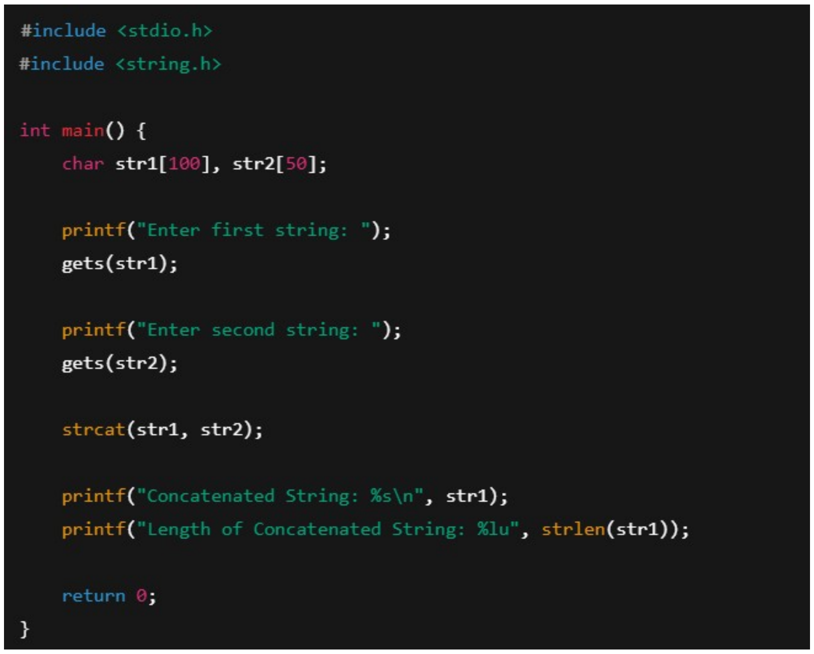
Character 'W' found in str3

*/

LAB EXERCISE:

Write a C program that takes two strings from the user and concatenates them using strcat(). Display the concatenated string and its length using strlen().

Ans:



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

int main() {
    char str1[100], str2[50];

    printf("Enter first string: ");
    gets(str1);

    printf("Enter second string: ");
    gets(str2);

    strcat(str1, str2);

    printf("Concatenated String: %s\n", str1);
    printf("Length of Concatenated String: %lu", strlen(str1));

    return 0;
}
```

/* Outputs

Enter first string: Hello

Enter second string: World

Concatenated String: HelloWorld

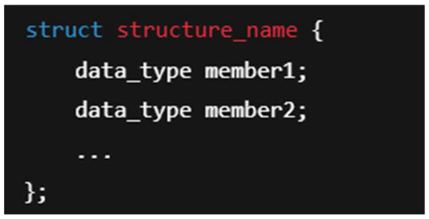
Length of Concatenated String: 10

*/

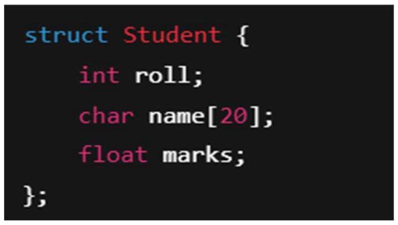
Q.12. Explain the concept of structures in C. Describe how to declare, initialize, and access structure members.

Ans: A **structure** in C is a **user-defined data type** that allows grouping of **different data types** under a single name. It is useful when we want to represent a real-world entity that has multiple attributes.

A structure is declared using the struct keyword.



```
struct structure_name {
    data_type member1;
    data_type member2;
    ...
};
```



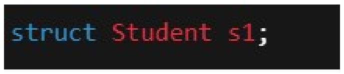
```
struct Student {
    int roll;
    char name[20];
    float marks;
};
```

Declaring Structure Variables:

Structure variables can be declared:

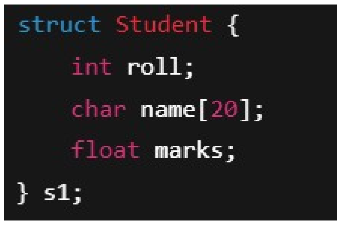
- After structure definition
- Along with structure definition

Method 1: After definition



```
struct Student s1;
```

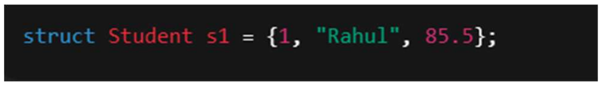
Method 2: Along with definition



```
struct Student {
    int roll;
    char name[20];
    float marks;
} s1;
```

Initialization of Structure:

Structure members can be initialized at the time of declaration.

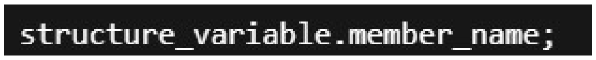


```
struct Student s1 = {1, "Rahul", 85.5};
```

Values must be given in the same order as structure members.

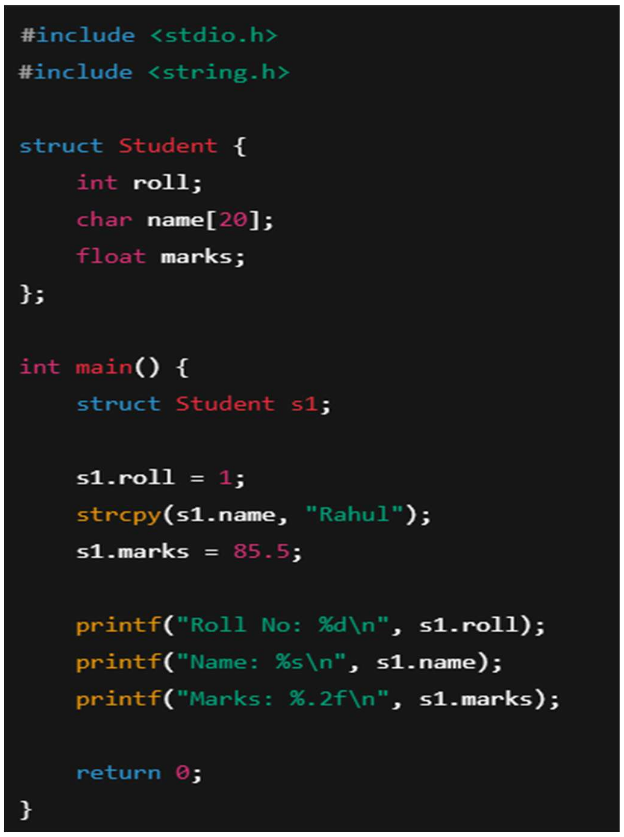
Accessing Structure Members:

Structure members are accessed using the **dot (.) operator**.



```
structure_variable.member_name;
```

Example:



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

struct Student {
    int roll;
    char name[20];
    float marks;
};

int main() {
    struct Student s1;

    s1.roll = 1;
    strcpy(s1.name, "Rahul");
    s1.marks = 85.5;

    printf("Roll No: %d\n", s1.roll);
    printf("Name: %s\n", s1.name);
    printf("Marks: %.2f\n", s1.marks);

    return 0;
}
```

```
/* Output

Roll No: 1

Name: Rahul

Marks: 85.50

 */
```

LAB EXERCISE:

Write a C program that defines a structure to store a student's details (name, roll number, and marks). Use an array of structures to store details of 3 students and print them.

Ans:

```c
#include <stdio.h>

struct Student {

    int roll;

    char name[20];

    float marks;
};

int main() {

    struct Student s[3];

    int i;


    /* Input student details */

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

        printf("\nEnter details of Student %d\n", i + 1);


        printf("Roll Number: ");

        scanf("%d", &s[i].roll);


        printf("Name: ");

        scanf("%s", s[i].name);


        printf("Marks: ");

        scanf("%f", &s[i].marks);

    }


    /* Display student details */

    printf("\n--- Student Details ---\n");

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

        printf("\nStudent %d\n", i + 1);

        printf("Roll Number: %d\n", s[i].roll);

        printf("Name: %s\n", s[i].name);

        printf("Marks: %.2f\n", s[i].marks);

    }


    return 0;

}
```

Outputs:

```
-----------------------------------------
Enter details of Student 1
Roll Number: 121
Name: mithlesh
Marks: 79

Enter details of Student 2
Roll Number: 122
Name: amit
Marks: 78

Enter details of Student 3
Roll Number: 123
Name: amar
Marks: 77

--- Student Details ---

Student 1
Roll Number: 121
Name: mithlesh
Marks: 79.00

Student 2
Roll Number: 122
Name: amit
Marks: 78.00

Student 3
Roll Number: 123
```

Name: amar

Marks: 77.00

Q.13. Explain the importance of file handling in C. Discuss how to perform file operations like opening, closing, reading, and writing files.

Ans: **Importance of File Handling in C**

File handling in C allows programs to **store data permanently** in files instead of losing it when the program terminates.

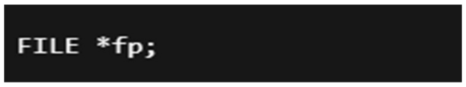
Why file handling is important:

- Stores data **permanently** (non-volatile storage)
- Helps in **large data management**
- Allows **data sharing** between programs
- Useful for **records, logs, reports, databases**
- Reduces memory usage by storing data externally

Files in C

A **file** is a collection of data stored on secondary storage (hard disk).

C uses a **FILE pointer** to work with files.

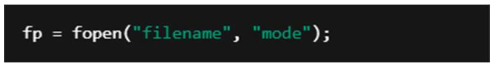


```
FILE *fp;
```

1. Opening a File

Files are opened using the fopen() function.

Syntax:



```
fp = fopen("filename", "mode");
```

Mode	Description
r	Read only
w	Write (creates new file)
a	Append

Mode	Description
r+	Read and write
w+	Write and read
a+	Append and read

Example:

```
fp = fopen("data.txt", "w");
```

2. **Closing a File**

After completing file operations, the file must be closed.

Syntax:

```
fclose(fp);
```

Why close a file:

- Saves data properly
- Frees memory
- Prevents data corruption

3. Writing to a File

Data can be written using functions like fprintf() or fputs().

Example (Writing to file):

```
fprintf(fp, "Hello File Handling");
```

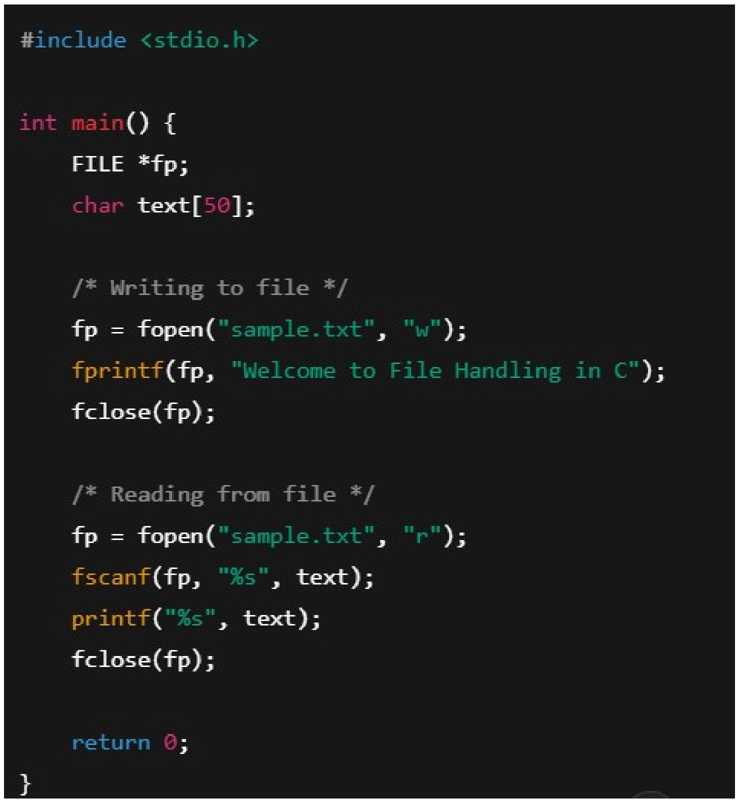
5. Reading from a File

Data can be read using functions like fscanf(), fgets(), or getc().

Example (Reading from file):

```
fscanf(fp, "%s", text);
```

Complete Example Program (Read & Write):



```c
#include <stdio.h>

int main() {
    FILE *fp;
    char text[50];

    /* Writing to file */
    fp = fopen("sample.txt", "w");
    fprintf(fp, "Welcome to File Handling in C");
    fclose(fp);

    /* Reading from file */
    fp = fopen("sample.txt", "r");
    fscanf(fp, "%s", text);
    printf("%s", text);
    fclose(fp);

    return 0;
}
```

Output:

```
Welcome
```

6. Key File Handling Functions:

Function	**Purpose**
fopen()	Opens a file
fclose()	Closes a file
fprintf()	Writes formatted data
fscanf()	Reads formatted data
fgets()	Reads a line
fputs()	Writes a line

LAB EXERCISE:

Write a C program to create a file, write a string into it, close the file, then open the file again to read and display its contents.

Ans:

```
#include <stdio.h>

int main() {

  FILE *fp;

  char text[100];


  /* Create and open file in write mode */

  fp = fopen("sample.txt", "w");

  if (fp == NULL) {

    printf("Error opening file!\n");

    return 1;

  }


  /* Write string into file */

  fprintf(fp, "Hello, this is file handling in C.");

  /* Close the file */

  fclose(fp);


  /* Open file in read mode */

  fp = fopen("sample.txt", "r");


  if (fp == NULL) {

    printf("Error opening file for reading! \n");

    return 1;

  }


  /* Read and display file content */

  printf("File Contents:\n");
```

```
     while (fgets(text, sizeof(text), fp) != NULL) {
      printf("%s", text);
  }

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

```
Output:
-----------------
File Contents:
Hello, this is file handling in C.
```