# Module 2 – Introduction to Programming

## 1. Overview of C Programming

## THEORY EXERCISE:
## Q. Write an essay covering the history and evolution of C programming. Explain its importance and why it is still used today.

→ C programming is a powerful and widely used programming language known for its

simplicity and versatility. Developed in the early 1970s by Dennis Ritchie at Bell Labs, C has since become one of the most influential programming languages in the world.

→ C became famous because it was used to build the **UNIX operating system**. Before that,

operating systems were written in assembly language, which was very hard. C made it easier, faster, and more powerful.

→ Evolution of C: -

In **1978**, **Brian Kernighan and Dennis Ritchie** wrote a book called *The C Programming Language*. This book made C even more popular.
Later, C was given official standards to keep it the same everywhere:
  • **ANSI C (1989)**
  • **C99 (1999)** • **C11 (2011)**
  • **C18 (2018)**

→ Importance of C
C is called the **mother of programming languages** because many languages like **C++, Java, and Python** came from it.

C is important because:
  • It is **fast** and uses **less memory**.
  • It is **portable** (same code works on different computers).
  • It helps programmers understand how computers really work.
  • It is used in **operating systems** (Windows, Linux, etc.).
  • It is used in **embedded systems** (mobiles, cars, machines).

→ **Why C is Still Used**

Even after 50 years, C is still used today because it is simple, powerful, and close to hardware. It is perfect for building software that needs **speed and control**.

# 1. Setting Up Environment

## Q. Describe the steps to install a C compiler (e.g., GCC) and set up an Integrated Development Environment (IDE) like Dec++, VS Code, or Code Blocks.

→ Step 1: Install GCC Compiler
- Download **MinGW** (Minimalist GNU for Windows) from its official website.
- Run the installer.
- In the setup, tick **"mingw32-gcc-g++"** (this is the C/C++ compiler).
- After installation, add MinGW's bin folder path (like C:\MinGW\bin) to **Environment Variables PATH**.

→ This step lets Windows find the compiler.
- o   To check if it works:
- o   Open **Command Prompt** o gcc –version → **Step 2:**

**Choose and Install an IDE**

Option 1: **DevC++**

- Download DevC++ from the official site.

- Install it like normal software (Next → Next → Finish).

- Open DevC++ → New Project → Console Application → Choose **C Language**.

- Write your C code → Press **F11** to run.

- DevC++ already comes with a compiler, so it's easiest for beginners.

→ Option 2: **Visual Studio Code (VS Code)**

- Download and install **VS Code**.

- Open VS Code → Go to Extensions → Install **C/C++ by Microsoft**.

- Make sure **MinGW (GCC)** is installed and added to PATH (Step 1).

- Create a new file hello.c → Write your C program. • Open **Terminal inside VS Code** → Run commands:

→ **Easiest way (for beginners):** Use **DevC++** because it comes with a built-in compiler. → **Better for projects:** Use **Code: Blocks** or **VS Code** for more features.

# 2. Basic Structure of a C Program

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

→ **Basic Structure of a C Program**

Every C program follows a standard structure. The main parts are:

- **Header Files**
- **Main Function**
- **Comments**
- **Data Types**
- **Variables 1) Header file: -**

☐ These are special files that contain built-in functions (like printf for output, scanf for input).

☐ Declared at the top using #include.

Example: -
#include <stdio.h> // Standard input-output header

### 2) Main Function: -

☐ Every C program must have a main () function.

☐ Program execution starts from here.

Example: - int
main () {
    // Code goes here
    return 0; // End of program
}

### 3) Comments: -

- Notes for programmers, ignored by the compiler.
- Two types:
    - ○ **Single-line:** // comment here ○
    - **Multi-line:** /* comment here */
- Example: -
    #include<stdio.h>
    Main()
    {
    // This is a single-line comment
    /* This is
       a multi-line comment */ return
    0;
    }

4) **Data Type: -**
- Data types are used to define the type of data that a variable can store.
- Basic Data type: -
  - int → integers (10, -5, 100)  ○ float →
  - decimal numbers (3.14, -2.5)  ○ char →
  - single character ('A', 'b')  ○ double →
  - larger decimal numbers

5) **Variables: -**
- Variable are fundamental elements used to store and manipulate data.
- They act as named container that hold different types of values, such as integer, floatingpoint numbers, characters, and pointer.

  Example: -
  int age = 20; // integer variable float pi =
  3.14;    // floating point variable char
  grade = 'A'; // character variable

Example: -

#include <stdio.h>   // Header file

// This program shows the basic structure of C int

main () {

   // Variable declaration

int age = 20;    float

height = 5.8;    char

grade = 'A';

   // Printing values

printf("Age: %d\n", age);

printf("Height: %.1f\n", height);

printf("Grade: %c\n", grade);

return 0;  // End of program

}

# 3. Operators in C

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

### Operator: -
Operators are **symbols** used to perform operations on values and variables.
Example: +, -, *, /

1. Arithmetic Operators

Used for **mathematical calculations**.

- + → addition
- - → subtraction
- * → multiplication
- / → division
- % → remainder (modulus)

Example: - #include<stdio.h>

Main() {

Int a=10, b=5;

Printf ("%d", a+b); //15

Printf ("%d", a-b); //5

Printf ("%d", a*b); //50

Printf ("%d", a/b); //2

Printf ("%d", a%b); //0

Return 0;

}

**2.** Relational Operators

Used to **compare two values**.

- == → equal to
- != → not equal to
- > → greater than
- < → less than
- >= → greater or equal
- <= → less or equal Example: - #include<stdio.h>

   Main() {

   Int a=10, b=5;

   Printf ("%d", a==b); //false

   Printf ("%d", a! =b); //true

   Printf ("%d", a>b); //true

   Printf ("%d", a<b); //false

   Printf ("%d", a>=b); //true

   Printf ("%d", a<=b); //false

   Return 0;

   }

**3.** Logical Operator

Used to combine **conditions**.

- `&&` → AND (true if both are true)
- `||` → OR (true if one is true)
- `!` → NOT (reverse the result)

  Example: - #include<stdio.h>
  Main()
  {
  int a = 5, b = 10; printf("%d", (a < b &&
  b > 0)); // 1 (true) printf("%d", !(a < b));
  // 0 (false)
  printf("%d", (a==5 || b==5); // 1 (true)
  }

## 4. **Assignment Operators**

Used to **store values** in variables.

- `=` → assign
- `+=` → add and assign
- `-=` → subtract and assign
- `*=` → multiply and assign
- `/=` → divide and assign Example: - #include <stdio.h> int main() {    int a;    // =

  (assign)    a = 10;

     printf("a = %d\n", a);   // Output: 10    //

  += (add and assign)    a += 5;  // a = a + 5

  → 10 + 5 = 15    printf("a += 5 → %d\n",

  a);   // Output: 15

// -= (subtract and assign)     a -= 3;   //

a = a - 3 → 15 - 3 = 12     printf("a -= 3

→ %d\n", a);   // Output: 12


// *= (multiply and assign)     a *= 2;   // a

= a * 2 → 12 * 2 = 24     printf("a *= 2

→ %d\n", a);   // Output: 24


// /= (divide and assign)     a /= 4;   // a

= a / 4 → 24 / 4 = 6     printf("a /= 4

→ %d\n", a);   // Output: 6


    return 0;

}


## 4. Increment / Decrement Operators

➤ Used to **increase or decrease** value by 1.

- **++** → increment
- -- → decrement There

are **two types**:

1. **Pre-increment / Pre-decrement** → Changes the value first, then uses it.
2. **Post-increment / Post-decrement** → Uses the value first, then changes it.

Example: - #include

<stdio.h>

```c
int main()
{    int a = 5,
b;

    // Pre-increment (++a)    b = ++a;   // a is
increased first, then assigned    printf("Pre-
increment: a = %d, b = %d\n", a, b);

    // Post-increment (a++)    b = a++;   // value of a is
assigned first, then increased    printf("Post-increment:
a = %d, b = %d\n", a, b);

    // Pre-decrement (--a)    b = --a;   // a is
decreased first, then assigned    printf("Pre-
decrement: a = %d, b = %d\n", a, b);

    // Post-decrement (a--)
    b = a--;   // value of a is assigned first, then decreased    printf("Post-
decrement: a = %d, b = %d\n", a, b);

    return 0;
}
```

Output: -

Pre-increment: a = 6, b = 6

Post-increment: a = 7, b = 6

Pre-decrement: a = 6, b = 6

Post-decrement: a = 5, b = 6

## 6. Bitwise Operators

Work at the **bit (0/1) level**.

- & → AND
- | → OR
- ^ → XOR (exclusive OR)
- ~ → NOT (flip bits)
- << → left shift
- >> → right shift Example: - int a = 5, b = 3; // 5 = 101, 3 = 011

(binary) printf("%d", a & b); // 1 (001) printf("%d", a | b); // 7 (111)

## 7. Conditional (Ternary) Operator

Shortcut for **if-else**.

- Syntax: condition? value_if_true: value_if_false

Example: - #include

<stdio.h>

int main() {     int num;

printf("Enter a number: ");

scanf("%d", &num);

```
    // Ternary operator (? :)

    (num % 2 == 0)

        ? printf("%d is Even\n", num)

        : printf("%d is Odd\n", num);


    return 0;

}
```

## 5. Control Flow Statements in C

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

1. **if statement**

  • Used when you want to run some code **only if a condition is true**.

    Syntax: - **if (condition) {**

      **// code runs if condition is true**

    **}**

Example: -

    #include <stdio.h> int main() {     int age = 20;     if (age >=

    18) {   // condition       printf("You are an Adult.\n");  // runs

    only if condition true

```
    }

    return 0;

}
```

## 2. if-else statement

• Used when you want to choose between **two options**.

Syntax: - if (condition) {

   // code runs if condition is true

} else {
   // code runs if condition is false

}


Example: -

#include

<stdio.h> int

main() {    int

num = 5;


   if (num % 2 == 0)

{       printf("Even number\n");

   } else {       printf("Odd

number\n");

   }

   return 0;

}

## 3. nested if-else

- Means using **if inside another if**. Used when there are **multiple conditions**.
  Syntax: - if (condition1) {
      // code if condition1 is true
    if (condition2) {
        // code if condition2 is also true
      } else {
        // code if condition2 is false
      }
    } else {
        // code if condition1 is false
    }

Example: -

```
#include <stdio.h> int

main() {

   int a = 10, b = 20, c = 15;


   if (a > b)
{      if (a > c)
{

        printf("a is the biggest\n");

     } else {

        printf("c is the biggest\n");

     }

   } else
{      if (b > c)
{

        printf("b is the biggest\n");

     } else {

        printf("c is the biggest\n");

     }

   }
```

```c
    return 0;

}
```

## 4. switch statement

- Used when you want to compare **one variable** with **many possible values** (instead of writing many ifelse).

```c
Syntax: - switch
(condition)
{    case value1:
// code        break;
case value2:
// code        break;
    ...
default:
        // code if no case matches
}
```

Example: -

```c
#include <stdio.h>

int main() {    int

choice;


    printf("Enter a number (1-3): ");

scanf("%d", &choice);


    switch (choice) {        case 1:

printf("You chose ONE\n");

break;        case 2:

printf("You chose TWO\n");

break;        case 3:

printf("You chose THREE\n");

break;        default:

        printf("Invalid choice!\n");

    }


    return 0;

}
```

- if → check 1 condition.  if-**else** →

- choose between 2 conditions.

- nested **if-else** → check multiple conditions step by step.

- switch → better when one variable has many possible values.

# 6. Looking in C

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

1. **while loop**

- Condition is checked **before** the loop runs. • If condition is
  **true**, loop runs. If false, it stops immediately.

  Syntax: - while
  (condition) {     //
  code to repeat
  }

  Example: -
  ```
  #include <stdio.h>
  int main() {     int i =
  1;     while (i <= 5)
  {       printf("%d\n",
  i);
       i++;
    }
    return 0;
  }
  ```

2. **for loop**

- Used when you **know exactly how many times** you want to repeat.
- Initialization, condition, and update are written in one line.

  Syntax: - for (initialization; condition;
  modification) {
     // code to repeat
  }

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

```
    for (int i = 1; i <= 5; i++)
{       printf("%d\n", i);
    }
    return 0;
}
```

## 3. do-while loop

• Runs the loop **at least once**, even if condition is false. •
Condition is checked **after** running the loop.

Syntax: - do
{
    // code to repeat
} while (condition);

Example: - #include
<stdio.h> int main()
{    int i = 1;    do
{       printf("%d\n",
i);       i++;    }
while (i <= 5);
return 0;
}

| Feature | while loop | for loop | do-while loop |
|---|---|---|---|
| **Condition check** | Before loop body | Before loop body | After loop body |
| **Guaranteed execution** | ✕ Not guaranteed | ✕ Not guaranteed | ✅ At least once |
| **Best for** | Unknown iterations | Known/finite iterations | Must run at least once |
| **Example use case** | Reading file until EOF | Iterating through array indexes | Menu system, input prompt |

# 7. Loop Control Statements

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

1. break – Stop the loop

- **Easy words:** If the loop is running, break will **stop it immediately**.

**Example:**

```
#include <stdio.h> int
main() {
    for (int i = 1; i <= 5; i++)
{       if (i == 3) {
            break;  // Stop the loop
        }
        printf("%d\n", i);
    }
    return 0;
}
```

Output: -

1

2

2. continue – Skip this turn

- **Easy words:** continue will **skip the current number** and go to the next one.

**Example:**

```c
#include <stdio.h> int main()
{    for (int i = 1; i <= 5; i++)
{        if (i == 3) {            continue;
// Skip number 3
        }
printf("%d\n", i);
    }
    return 0;
}
```

Output: -

1

2

4

5

# 3. goto – Jump to a label

- **Easy words:** goto lets you **jump to a specific part of the program**.

**Example:**

```c
#include    <stdio.h>
int main() {     int i =
1; start:  // label     if
(i        <=        3)
{        printf("%d\n",
i);      i++;        goto
```

```
start;   // Jump  back
```

to "start"

```
    }

    return 0;

}
```

# 8. Functions in C

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

- A **function** in C is a **block of code** that performs a specific task.
  It helps to **divide a big program into smaller parts**, making the program **easy to understand and reuse**.

There are **3 main parts** of a function:

**1. Function Declaration (Prototype)**

- Tells the compiler about the function name, return type, and parameters
(if any).
- It's usually written **before the main()** function. Syntax: - return_type

function_name(data_type parameter1, data_type parameter2); Example: -

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

**2. Function Definition**

- This is where we **actually write the code** of the function — what it will

do. Syntax: - return_type function_name(data_type parameter1, data_type

parameter2) {

```
    // code to perform task
```

return value;

```
}
```
Example: -

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

{    int sum = a + b;

return sum;

}
```

## 3. Function Call

- To **use** the function, we call it inside the main() function. Syntax: - function_name(arguments); Example: - int result = add(5, 3); example: -

#include <stdio.h>

```
// Function Declaration int
add(int a, int b);


int main() {    int
x, y, result;    x
= 5;    y = 3;


   //    Function    Call
result   =   add(x,   y);
printf("Sum    =    %d",
result);    return 0;

}


// Function Definition
int add(int a, int b)
```

```
{    int sum = a + b;

return sum;

}
```

# 9. Arrays in C

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

• An **array** is a **collection of similar data items** stored at **continuous memory locations**. It is used to **store multiple values** of the **same data type** using a **single name**. Example: - int marks[5];

## There are two main types:

**1.** One-Dimensional Array

- It stores data in a **single row (line)**.
- Syntax: - data_type array_name[size];

**2.** Multi-Dimensional Array

- It stores data in **rows and columns** (like a table).The most common is the **two-dimensional array**.
- Syntax: - **data_type array_name[rows][columns];**

| Feature | One-Dimensional Array | Multi-Dimensional Array |
|---------|----------------------|-------------------------|
| Structure | Single row (line) | Table-like (rows & columns) |
| Syntax | int a[5]; | int a[2][3]; |
| Example | int marks[5] = {10, 20, 30, 40, 50}; | int mat[2][3] = {{1,2,3},{4,5,6}}; |
| Access element | marks[2] | mat[1][2] |
| Use | Store list of items | Store data in matrix or table form |

# 10. Pointers in C

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

- A **pointer** is a **special variable** that **stores the address of another variable**. Instead of storing a value directly, it stores **where the value is kept in memory**.
- A **variable** store a value.
- A **pointer** stores the **address** of that variable. Syntax: - data_type *pointer_name;

Example: - int a = 10;     // normal variable int *p;        // pointer variable p = &a;        // store address of 'a' in pointer 'p'

Example: - #include

<stdio.h>

int main() {    int a = 10;    int *p;

// pointer declaration    p = &a;

// pointer initialization

```
printf("Value of a: %d\n", a);
printf("Address of a: %p\n", &a);
printf("Pointer p stores: %p\n", p);
printf("Value pointed by p: %d\n", *p);

return 0;
}
```

Output: - Value

of a: 10

Address of a: 0x7fffb2b3a5c

Pointer p stores: 0x7fffb2b3a5c

Value pointed by p: 10

| Reason | Explanation |
|---|---|
| 1 Access memory directly | You can read/write data using memory addresses |
| 2 Dynamic memory | Used in malloc(), calloc(), and free() |
| 3 Faster and efficient | Helps pass large data (like arrays) without copying |
| 4 Function arguments | Functions can modify variables outside their scope |
| 5 Data structures | Used in linked lists, trees, stacks, queues, etc. |

# 11. Strings in C

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

 • A **string** in C is a **collection of characters** stored in an array and **ends with a null character** ('\0').

Example: - char name[] = "Rana";

Here, the string "Rana" is actually stored as:
'R' 'a' 'n' 'a' '\0'

## Common String Handling Functions (in <string.h>)

These functions are built into C for working with strings. You
must include the header file:

#include <string.h>

## 1. strlen() – Find Length of a String

 • **Use:** Returns the **number of characters** in a string

(excluding '\0'). Syntax: - int strlen(char str[]);

## 2. strcpy() – Copy One String to Another

- **Use:** Copies the content of one string into another.

Syntax: -

strcpy(destination, source);

## 3. strcat() – Join Two Strings

- **Use:** Combines (concatenates) two strings together.

Syntax: -

strcat(string1, string2);

## 4. strcmp() – Compare Two Strings

- **Use:** Compares two strings **character by character**.

Syntax: - strcmp(string1, string2);

## 5. strchr() – Find Character in a String

- **Use:** Finds the **first occurrence** of a character in a string. Syntax: - strchr(string, character);

| Function | Meaning | Example | Use |
|----------|---------|---------|-----|
| strlen() | Finds string length | strlen("Hello") → 5 | Count characters |
| strcpy() | Copies one string to another | strcpy(b, a) | Copy names, messages |
| strcat() | Joins two strings | strcat(a, b) | Combine text |
| strcmp() | Compares two strings | strcmp(a, b) | Compare input (like passwords) |
| strchr() | Finds a character | strchr("apple",'p') | Search characters |

Example: - #include

<stdio.h>

#include <string.h>

int main() {     char

str1[50] = "Hello";     char

str2[50] = "World";     char

```c
    str3[50];    char *ptr;
    int len, cmp;

    // 1  strlen() - Find length of string
    len = strlen(str1);    printf("Length of
str1 = %d\n", len);

    // 2  strcpy() - Copy str1 into str3
    strcpy(str3, str1);    printf("After strcpy,
str3 = %s\n", str3);

    // 3  strcat() - Join str1 and str2
    strcat(str1, str2);    printf("After
strcat, str1 = %s\n", str1);

    // 4  strcmp() - Compare str1 and
str2    cmp = strcmp(str1, str2);
    if(cmp == 0)        printf("str1 and str2
are equal\n");    else if(cmp > 0)
printf("str1 is greater than str2\n");
    else        printf("str1 is smaller than
str2\n");

    // 5  strchr() - Find character in string    ptr = strchr(str1, 'o');
    if(ptr)        printf("Character 'o' found at position: %ld\n", ptr -
str1 + 1);    else        printf("Character not found\n");

    return 0;
}
```

# 12. Structures in C

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

- A **structure** is a **user-defined data type** that lets you **combine different types of data** under one name.
  It's like a box that can hold **different types of variables** together.

  Think of it like this:
  If you want to store a student's details — name, roll number, and marks — you can use a **structure** instead of separate variables.

☐ To **group related information** of different data types.

☐ Makes code cleaner and easier to manage.

☐ Useful for **real-world data** (like student, employee, book, etc.).

## 1. Declaring a Structure

**Synatx:** - struct

structure_name

{ data_type

member1; data_type

member2;

...

};

Example: - struct

Student {

int roll;

char name[50];

float marks;

};

## 2. Declaring Structure Variables struct Student s1, s2;

## 3. Initializing a Structure

Example: - struct

Student          s2;

s2.roll = 2;

strcpy(s2.name, "Vrujal");   // use strcpy() for strings s2.marks

= 92.0;

## 4. Accessing Structure Members

Use the **dot operator (.)** to access members. Example:

-

```c
printf("Roll: %d\n", s1.roll);
printf("Name: %s\n", s1.name);

printf("Marks: %.2f\n", s1.marks);
```

Example: - #include

<stdio.h>

#include <string.h>

```c
// Structure Declaration struct
Student {
    int roll;
    char name[50];
float marks;
};

int main() {
    // Structure Initialization     struct
Student s1 = {1, "Rana", 88.5};

    // Another way to assign values
struct Student s2;     s2.roll = 2;
```

```c
strcpy(s2.name, "Vrujal");

s2.marks = 92.0;


    // Displaying Data
printf("Student 1:\n");

    printf(" Roll: %d\n Name: %s\n Marks: %.2f\n\n", s1.roll, s1.name, s1.marks);


    printf("Student 2:\n");     printf(" Roll: %d\n Name: %s\n Marks: %.2f\n",

s2.roll, s2.name, s2.marks);     return 0;


}
```

13. File Handling in C

Q. 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 allows you to **store data permanently** on your computer (in files), so that data is **not lost when the program ends**.

Without file handling — data disappears when the program closes.
With file handling — data can be **saved, read, or modified** anytime.

| Reason | Explanation |
|---|---|
| ①  Permanent storage | Keeps data safe even after program ends |
| ②  Data sharing | Files can be shared between programs |
| ③  Easy access | Read/write large data easily |
| ④  Record keeping | Used in databases, reports, and logs |


All file handling functions are found in the **<stdio.h>** library.

| Function | Purpose |
|---|---|
| fopen() | Opens a file |
| fclose() | Closes a file |
| fprintf() | Writes formatted data to a file |
| fscanf() | Reads formatted data from a file |
| fgets() | Reads a line from a file |
| fputs() | Writes a line to a file |

| | |
|---|---|
| **fgetc()** | Reads a character |
| **fputc()** | Writes a character |

1. Declare a File Pointer

FILE *fp;

2. Open a File using fopen() fp =

   fopen("data.txt", "w");

| Mode | Meaning | Description |
|---|---|---|
| **"r"** | Read | Opens existing file for reading |
| **"w"** | Write | Creates new file (deletes old one) |
| **"a"** | Append | Adds data at the end of file |
| **"r+"** | Read + Write | Opens existing file for both |
| **"w+"** | Write + Read | Creates new file for both |

3. Perform Read/Write Operations

   4. Close the File using fclose()

   fclose(fp);

Example 1: Writing to a File

```c
#include <stdio.h>

int main() {    FILE *fp;    fp = fopen("example.txt",
"w");  // open for writing

   if(fp == NULL)
{        printf("Error opening
file!");        return 1;
   }
```

```c
    fprintf(fp, "Hello, this is a test file.\n");
fprintf(fp, "Writing data using fprintf().");


    fclose(fp); // close file    printf("Data
written successfully!\n");    return 0;

}
```

**Output: -**

Data written successfully!

# Content in "example.txt":

Hello, this is a test file.

Writing data using fprintf().


Example 2: Reading from a File

```c
#include <stdio.h>

int main()
{    FILE *fp;
char ch;


    fp = fopen("example.txt", "r"); // open for reading
if(fp == NULL) {        printf("File not found!");
return 1;
    }


    printf("File content:\n");
while((ch = fgetc(fp)) != EOF)
{        printf("%c", ch);
    }


    fclose(fp);
return 0;
```

```
}
```

**Output:**

File content:

Hello, this is a test file.

Writing data using fprintf().