Software Engineering Assignment

MODULE: 2

Introduction to Programming

---

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

=> The C programming language is a foundational language in computer science, known for its efficiency, portability, and influence on subsequent programming languages. Created in the early 1970s by Dennis Ritchie at Bell Labs, C emerged as a powerful tool for system programming, initially designed to develop the UNIX operating system.

Origins of C:

C originated as an improvement on the B language, which itself was derived from BCPL, a language used for system programming. B had limitations in handling data types and performance, prompting Ritchie to create C between 1969 and 1973. Its powerful handling of memory and data types enabled UNIX to be written in C, making it portable across various hardware, an innovative feature at the time.

Influence on Other Languages:

C's syntax and design influenced many popular languages, including C++, Java, C#, and Python. It introduced core principles like portability and low-level memory management, which continue to define systems programming. C++ was developed as an extension of C, while Java and C# adopted C's syntax with additional memory management features suited to application development.

Conclusion:

C's evolution, from a tool for creating UNIX to a standardized language used worldwide, underscores its adaptability and foundational impact. Its efficient design and influence across programming languages ensure that C remains relevant, secure, and essential in modern computing.

Importance of C and Its Continued Relevance Today:

The C programming language remains vital in the field of computing, not only for its historic contributions but also for its ongoing role in modern software development. Known for its efficiency, portability, and control over hardware resources, C has shaped the foundation of software engineering and system-level programming for decades. Here's why C continues to be essential

Low-Level Access to Hardware:

C allows direct access to memory and hardware through pointers, memory management, and bitwise operations, making it ideal for systems programming. This level of control is crucial for developing operating systems, device drivers, and embedded systems where fine-tuning memory and performance is essential.

Portability Across Platforms:

One of C's core design goals was portability, which has made it possible to write software that can run on various hardware platforms with minimal modification. C code can be compiled and executed on different machines, making it invaluable for cross-platform development. This portability also allowed C to become the language of choice for creating operating systems like UNIX, which later influenced operating systems like Linux, macOS, and Windows.

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.

=> Steps to Install a C Compiler (GCC) and Set Up an IDE (DevC++, VS Code, or Code::Blocks)

To begin programming in C, you'll need to install a compiler, such as GCC (GNU Compiler Collection), and set up an Integrated Development Environment (IDE) to streamline the coding process. Below are steps for installing GCC and setting up popular IDEs: DevC++, Visual Studio Code (VS Code), and Code::Blocks.

Step 1: Install a C Compiler (GCC)

1.Download MinGW (Minimalist GNU for Windows), which includes GCC:

-Visit the MinGW website or MSYS2 website for a newer option.

-Download and install the MinGW or MSYS2 installer.

i)Install GCC via MinGW or MSYS2:

-MinGW: Run the mingw-get-setup.exe file. Open the installer, select "mingw32-gcc-g++" in the package list, and install it.

ii)Add GCC to Path:

-Go to **Control Panel > System and Security > System > Advanced System Settings > Environment Variables**.

-Find the **Path** variable under **System Variables**, click **Edit**, and add the path to the GCC binary (e.g., C:\MinGW\bin or the MSYS2 equivalent).

-Restart your terminal to make sure the PATH changes take effect.

Step 2: Set Up an IDE

Once GCC is installed, you can set up an IDE to make coding easier with features like syntax highlighting, debugging, and project management.

DevC++(Windows):

i)Download DevC++:

-Go to the DevC++ website and download the installer.

ii)Install DevC++:

-Run the installer and follow the on-screen instructions to install DevC++.

iii)Configure DevC++:

-Open DevC++, go to **Tools > Compiler Options** to verify that it detects GCC automatically.

-If not, specify the path to your GCC compiler (usually in C:\MinGW\bin if installed with MinGW).

Visual Studio Code (Cross-Platform):

i)Download VS Code:

-Go to the Visual Studio Code website and download the installer for your operating system.

ii)Install the C/C++ Extension:

-Open VS Code, go to the **Extensions** tab (or press Ctrl+Shift+X), and search for **C/C++** by Microsoft.

-Click **Install** to add the extension, which provides debugging and IntelliSense (code autocompletion) for C/C++.

iii)Configure Compiler and Debugger:

-Open VS Code, create a new file with a .c extension, and write a simple C program.

-Go to **Terminal > New Terminal** and verify that gcc is recognized by running gcc --version.

-Configure the **tasks.json** file to run the gcc compiler:

-Open **Run > Add Configuration…** and select C++: g++ build and debug active file. Update the path if needed.

-Configure the **launch.json** file for debugging:

-Open **Run > Add Configuration…** and choose **C++ (GDB/LLDB)**, then customize for your OS.

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

=> A C program has a simple structure, which includes headers, the main function, comments, data types, and variables. Each of these components plays a specific role in the program's organization, readability, and functionality. Below is a breakdown of each element with explanations and examples.

1. Headers

Headers are files that contain declarations of functions, macros, and types that a program can use. They are included at the beginning of a program using the #include directive. Commonly used headers in C include:

-<stdio.h> for input/output functions like printf and scanf

-<stdlib.h> for general utilities like memory allocation and random numbers

-<string.h> for string handling functions

Example:

```
#include <stdio.h>  // Standard Input/Output library
#include <stdlib.h> // Standard library functions
```

2. Main Function

Every C program must have a main function, which serves as the program's entry point. When a C program is executed, it begins running code from the main function. The main function typically has the following structure:

-Syntax: int main(void) or int main(int argc, char *argv[])

-Return Type: The main function returns an integer, with 0 typically signifying successful execution.

-Body: Enclosed in braces {}, it contains the code to be executed.

Example:

```
int main()
{
   // Code execution starts here
   printf("Hello, World!\n");
```

```
    return 0;  // Returning 0 indicates successful execution
}
```

3. Comments

Comments are lines that are ignored by the compiler. They are used to explain code and make it more readable. C supports two types of comments:

-Single-line comments start with //.

-Multi-line comments start with /* and end with */.

Example:

```
// This is a single-line comment

/*
 This is a
 multi-line comment
 explaining code
*/
```

4. Data Types

Data types specify the type of data that a variable can store. In C, each variable must have a declared data type. Common data types include:

-int: Integer numbers (e.g., 5, -3)

-float: Floating-point numbers (e.g., 3.14, -2.5)

-double: Double-precision floating-point numbers, for more accuracy than float

-char: Character data (e.g., 'A', 'z')

-void: Represents no data or “empty type”

Example:

```
int age = 25;          // Integer
float height = 5.9;    // Floating-point number
double distance = 10.5; // Double precision floating-point
char grade = 'A';      // Character
```

5. Variables

Variables are containers for storing data values. Each variable in C must be declared with a specific data type and assigned a name that follows naming conventions. The basic syntax for declaring a variable is:

data_type variable_name = value;

Variable names should be descriptive and are case-sensitive. They can consist of letters, digits, and underscores but cannot start with a digit.

Example:

```
int age = 25;          // Declaring an integer variable
float weight = 72.5;  // Declaring a floating-point variable
char initial = 'J';   // Declaring a character variable
```

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

=> Arithmetic Operators

Arithmetic operators perform basic mathematical operations.

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

2. Relational Operators

Relational operators compare two values and return a boolean result (1 for true, 0 for false).

Operator	Description	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

Logical operators are used to combine or invert boolean expressions, often with conditional statements.

Operator	Description	Example
&&	AND	a>0 && b>0
||	OR	a>0 || b>0
!	NOT	!(a>0)

4. Assignment Operators

Assignment operators assign values to variables. The simple = operator is used for assignment, and there are compound operators that combine assignment with arithmetic.

Operator	Description	Example
=	Simple assignmet	a=3
+=	Add and assign	a +=5
-=	Subtract and assign	a-=5
*=	Multiply and assign	a*=3
/=	Divide and assign	a/=5
%=	Modulus and assign	a%=8

5. Increment and Decrement Operators

These operators increase or decrease the value of a variable by 1.

Operator	Description	Example
++	Increment	++a or a++
--	Decrement	--a or a--

6. Conditional (Ternary) Operator

The conditional (ternary) operator is a compact way to evaluate an expression and return one of two values.

Syntax	Description	Example
?:	If-else shorthand operator	(a > b) ? a : b

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

=> In C, decision-making statements control the flow of execution based on certain conditions. Here's an explanation of the main decision-making statements with examples:

1. if Statement

The if statement allows code to execute only when a specified condition is true.

Syntax:

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

Example:

```
int num = 10;
if (num > 0)
{
    printf("The number is positive.\n");
}
```

In this example, if num is greater than 0, it will print "The number is positive."

2. else Statement

The `else` statement provides an alternative path if the condition in false.

Syntax:

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

```
}
```

Example:

```
int num = -5;
if (num > 0)
{
    printf("The number is positive.\n");
} else {
    printf("The number is non-positive.\n");
}
```

Here, if num is greater than 0, it will print "The number is positive." Otherwise, it will print "The number is non-positive."

3. Nested if-else Statement

In nested if-else, one if-else statement is placed inside another if or else block. This is useful for checking multiple conditions with hierarchy.

Syntax:

```
if (condition1) {
   // Outer if block
   if (condition2) {
      // Inner if block
   } else {
      // Inner else block
   }
} else {
   // Outer else block
}
```

Example:

```
int num = 15;
if (num > 0) {
    if (num % 2 == 0) {
        printf("The number is positive and even.\n");
    } else {
        printf("The number is positive and odd.\n");
        }
  } else {
    printf("The number is non-positive.\n");
  }
```

Here, the outer if checks if num is positive, and the inner if-else checks if num is even or odd.

5. switch Statement

The switch statement tests the value of an expression against multiple case values. It's often used for menus or when there are multiple fixed values for a single variable.

Syntax:

```
switch (expression) {
    case value1:
        // Code for case value1
        break;
    case value2:
        // Code for case value2
        break;
    ...
    default:
        // Code if none of the cases match
}
```

break: Exits the switch after executing a case. Without break, the program continues to execute the next case ("fall-through").

default: Executes if none of the cases match the expression value. It's optional but useful for handling unexpected values.

Example:

```
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");
}
```

In this example, since `day` is `3`, it will print "Wednesday." The `break` statement prevents other cases from executing after the matching case is found.

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

=> Comparison of While, For, and Do-While Loops:

Comparison of Loop Types:

Feature	While Loop	For Loop	Do-While Loop
Syntax	`while (condition) { ... }`	`for (initialization; condition; update) { ... }`	`do { ... } while (condition);`
Condition Check	At the beginning of the loop	At the beginning of the loop	At the end of the loop
Guaranteed Execution	Not guaranteed (may not run at all)	Not guaranteed (may not run at all)	Executes at least once
Best for	Repeating code while a condition is true	Iterating over a known range or collection	Running code that must execute at least once
Complexity Handling	Simpler for unknown termination logic	Well-suited for definite iterations	Simpler for post-condition checks

Use Cases for Each Loop

1.While Loop

Best Scenari : When the number of iterations is not predetermined, but the loop depends on a dynamic condition.

Examples

- Reading data from a file until EOF.
- Continuously prompting a user until valid input is received.
- Running an operation until a certain computational accuracy is achieved.

Example (C-like pseudocode):

```c
while (userInput != valid) {
    userInput = getInput();
}
```

2.For Loop

Best Scenario: When the number of iterations is known beforehand or when iterating over a range, collection, or sequence.

Examples

- Iterating over an array or list.
- Performing a calculation over a fixed range (e.g., summing numbers from 1 to 100).
- Repeating a block of code a specific number of times.

Example(C-like pseudocode):

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

3.Do-While Loop

Best Scenario: When the code block must execute at least once before checking the condition.

Examples

- Presenting a menu to a user and ensuring the menu is shown at least once before an exit option is checked.
- Running an operation that must execute initially regardless of condition validity.

Example (C-like pseudocode):

```c
do {
    userInput = getInput();
} while (userInput != valid);
```

7)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 mechanisms used to alter the normal flow of execution in loops and other constructs. Here's an explanation of each, with examples:

Break Statement:

-To break the code, rest of the code will not be executed.

Example:

```
#include <stdio.h>
 main()
{
    for (int i = 0; i < 10; i++)
   {
        if (i == 5)

{
            break;  // Exit the loop when i equals 5
        }
        printf("%d ", i);
    }
}
```

Continue Statement:

-To skip the current iteration. Rest of the code will be executed then.

Example:

```
#include <stdio.h>
main()
```

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

Goto Statement:

- Goto that label to continue with the code execution.

Example:

```
#include <stdio.h>
main()
{
    int num = 5;
     if (num < 10)
   {
        goto skip;  // Jump to the label 'skip'
    }
    printf("This will be skipped.\n");
```

```
skip:
   printf("This is the target of the goto.\n")
}
```

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

=> Functions in C:

Functions are blocks of code designed to perform a specific task. They promote code reuse, modularity, and readability. Functions in C can be built-in (e.g., `printf`) or user-defined.

Components of a Function:

1)Function Declaration (Prototype)

A function must be declared before it is called. The declaration specifies the function's name, return type, and parameters.

Example:

int a, b;

2)Function Definition:

The function definition provides the actual implementation (code) of the function. It includes the logic or task that the function performs.

Example:

Int a, b;

{

```
   Int a + b
}
```

3)Function Call

A function is executed by calling it from another part of the program. The caller passes arguments that match the function's parameter list.

Example:

```
int result = add(5, 10);
 // Call the 'add' function with arguments 5 and 10
```

Example: Complete Program with Function:

```
#include <stdio.h>
main() {
    int num1 = 5, num2 = 10
    int sum = add(num1, num2);
   printf("Sum of %d and %d is %d\n", num1, num2, sum);
}
```

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

=> Concept of Array in C:

An **array** is a collection of elements of the same data type stored in contiguous memory locations. Arrays allow you to group and process related data efficiently using a single variable name and an index.

Features of Arrays:

- Homogeneous: All elements in the array must be of the same type.

- Fixed Size: The size of the array is defined when it is declared and cannot be changed later.
- Indexed Access: Array elements are accessed using an index, starting from `0`.

Types of Arrays:

One-Dimensional Array:

A one-dimensional (1D) array is a linear collection of elements, often referred to as a list.

Exampel:

```
#include <stdio.h>

main()

{

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

   int sum = 0;


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

  {

      sum += numbers[i];

   }

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


}
```

Multi-Dimensional Array:

A multi-dimensional array is an array of arrays, where each element can itself be an array. The most common example is a **two-dimensional (2D) array**, often visualized as a matrix..

Example:

```
#include <stdio.h>
 main()
{
   int matrix[2][3] =
  {
      {1, 2, 3},
      {4, 5, 6}
   };
   for (int i = 0; i < 2; i++)

  {
      for (int j = 0; j < 3; j++)
     {
         printf("%d ", matrix[i][j]);
      }
      printf("\n");
  }
```

}

Key Differences Between 1D and Multi-Dimensional Arrays:

Feature	One-Dimensional Array	Multi-Dimensional Array
Structure	Linear (list-like)	Tabular or matrix-like
Declaration	data_type array[size]	data_type array[rows][columns]
Visualization	Single row of elements	Multiple rows and columns
Access	array[index]	array[row][column]
Example	int arr[5];	int mat[2][3];

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

=> Pointers in C:

A **pointer** is a variable in C that stores the memory address of another variable. Pointers provide powerful features like dynamic memory management, efficient data handling, and the ability to directly manipulate memory.

Key Features of Pointers:

1)Storage of Addresses: Pointers hold the address of a variable rather than its value.

2)Indirect Access: Using pointers, you can access or modify the value stored at the memory address they point to.
3)Dynamic Memory: Pointers enable the allocation and deallocation of memory at runtime.

=>Declaration:

Pointers are declared using the * operator before the pointer name.

Example:

```
int *p;  // Declares a pointer to an integer
float *q;  // Declares a pointer to a float
```

=>Initialization:

Pointers are initialized with the address of a variable using the address-of operator (&).

Example:

```
int x = 10;  // A normal integer variable
int *p = &x;  // Pointer 'p' stores the address of 'x'
```

Pointers Important in C:

i)Efficient Memory Access

Pointers provide direct access to memory, making operations faster in certain scenarios.

ii)Dynamic Memory Allocation
Functions like `malloc` and `free` use pointers to allocate and deallocate memory dynamically.

iii)Data Structures
Pointers are crucial for implementing complex data structures like linked lists, trees, and graphs.

iv)Function Arguments
Pointers allow functions to modify the actual variables passed to them, enabling pass-by-reference functionality.

vi)Pointer Arithmetic
Pointers can be incremented or decremented to traverse arrays or memory blocks.

vii)Low-Level Programming
Pointers enable direct memory manipulation, which is crucial in embedded systems, operating systems, and device drivers.

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

=> String Handling Functions in C:
C strings are arrays of characters terminated by a null character (`\0`). The standard library `<string.h>` provides several functions to perform

operations on strings. Below is an explanation of key string functions and examples.

1)`strlen()` – String Length:

- **Purpose**: Determines the length of a string, excluding the null terminator (`\0`).
- **Prototype**: `size_t strlen(const char *str);`
- **Returns**: The number of characters in the string.

Example:

```
#include <stdio.h>
#include <string.h>
 main()
{
   char str[] = "Hello, World!";
   printf("Length of the string: %lu\n", strlen(str));
}
```

2)`strcpy()` – String Copy:

- **Purpose**: Copies the contents of one string to another.
- **Prototype**: `char *strcpy(char *dest, const char *src);`
- **Returns**: A pointer to the destination string (`dest`).

Example:

```
#include <stdio.h>
#include <string.h>
main()
{
```

```
    char src[] = "Hello, C!";
    char dest[20];

    strcpy(dest, src);  // Copy the contents of 'src' into 'dest'
    printf("Copied string: %s\n", dest);
}
```

3)`strcat()` – String Concatenation:

- **Purpose**: Appends (concatenates) one string to the end of another.
- **Prototype**: `char *strcat(char *dest, const char *src);`
- **Returns**: A pointer to the destination string (`dest`).

Example:

```
#include <stdio.h>
#include <string.h>
main()
{
    char str1[20] = "Hello, ";
    char str2[] = "World!";

    strcat(str1, str2);  // Concatenate 'str2' to 'str1'
    printf("Concatenated string: %s\n", str1);

}
```

4)`strcmp()` – String Comparison:

- **Purpose**: Compares two strings lexicographically.
- **Prototype**: `int strcmp(const char *str1, const char *str2);`

- **Returns**:
- `0` if the strings are equal.
- `<0` if `str1` is lexicographically less than `str2`.
- `>0` if `str1` is lexicographically greater than `str2`.

Example:

```
#include <stdio.h>
#include <string.h>
main()
{
   char str1[] = "apple";
   char str2[] = "banana";

   int result = strcmp(str1, str2);

   if (result == 0)
   {
      printf("Strings are equal\n");
   } else if (result < 0)
   {
      printf("'%s' comes before '%s'\n", str1, str2);
   } else
   {
```

```
        printf("'%s' comes after '%s'\n", str1, str2);
    }
}
```

5)`strchr()` – Find Character in String:

- **Purpose**: Searches for the first occurrence of a character in a string.
- **Prototype**: `char *strchr(const char *str, int c);`
- **Returns**:
- A pointer to the first occurrence of the character.
- `NULL` if the character is not found.

Example:

```
#include <stdio.h>
#include <string.h>
 main()
{
    char str[] = "Hello, World!";
    char ch = 'o';

    char *pos = strchr(str, ch);

    if (pos != NULL)
   {
        printf("Character '%c' found at position: %ld\n", ch, pos - str);
```

```
    } else
   {
        printf("Character '%c' not found\n", ch);
    }
}
```

Function	Purpose	Return Value
strlen()	Finds the length of a string.	Number of characters (excluding `\0`).
strcpy()	Copies one string to another.	Pointer to the destination string.
strcat()	Concatenates two strings.	Pointer to the destination string.
strcmp()	Compares two strings lexicographically.	Integer indicating comparison result.
strchr()	Finds the first occurrence of a character.	Pointer to the character or `NULL`.

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

=> Concept of Structures in C:

A **structure** in C is a user-defined data type that groups related variables (of potentially different data types) under a single name. Structures help in organizing and handling complex data in a more manageable way.

Defining a Structure:

To define a structure, use the `struct` keyword followed by a name and a block containing the member variables.

Example:

```
struct Person
{
   char name[50];
   int age;
   float height;
};
```

Here, `Person` is a structure with three members: `name`, `age`, and `height`.

i)Declaring Structure Variables:

Once a structure is defined, you can declare variables of that type.

Example:

```
struct Person person1, person2;
```

ii)Initializing Structure Members:

Structure members can be initialized at the time of declaration or later.

Example:

1. Initialization at Declaration

```
struct Person person1 = {"John Doe", 25, 5.9};
```

2. Initialization After Declaration

```
struct Person person1;
strcpy(person1.name, "John Doe");
person1.age = 25;
```

person1.height = 5.9;

iii)Accessing Structure Members:

The **dot operator (.)** is used to access individual members of a structure.

Example:

```
#include <stdio.h>
#include <string.h>
 struct Person
{
   char name[50];
   int age;
   float height;
};

main()
{
   struct Person person1;

   // Initializing structure members
   strcpy(person1.name, "Alice");
   person1.age = 30;
   person1.height = 5.7;
```

```
    // Accessing structure members
    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);
    printf("Height: %.1f\n", person1.height);
}
```

13) 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

File handling in C allows programs to store and retrieve data on disk, making it persistent even after the program terminates. This capability is crucial for applications like databases, configuration storage, and logging.

C provides a set of functions in the **`<stdio.h>`** library to handle files.

Importance of File Handling

1. **Persistence**: Data can be saved to and retrieved from files for later use.
2. **Efficiency**: Large datasets can be processed directly from files, avoiding memory limitations.
3. **Data Sharing**: Files enable data exchange between applications and systems.
4. **Backup and Logs**: Files allow storing logs and backups for debugging and recovery.

File Operations:

The key operations in file handling are:

1. **Opening a file**: Using `fopen()`.
2. **Closing a file**: Using `fclose()`.
3. **Reading from a file**: Using `fgetc()`, `fgets()`, or `fread()`.
4. **Writing to a file**: Using `fputc()`, `fputs()`, or `fwrite`

1) Opening a File

Files must be opened before any operation. Use the `fopen()` function.

Example:

```
FILE *file = fopen("example.txt", "w");
if (file == NULL)
{
   printf("Error opening file!\n");
}
```

Modes:

Mode	Description
“r”	Opens a file for reading. File must exist.
“w”	Opens a file for writing. Overwrites if the file exists.
“a”	Opens a file for appending. Creates if it doesn’t exist.
“r+”	Opens a file for both reading and writing.
“w+”	Opens a file for both reading and writing. Overwrites.
“a+”	Opens a file for both reading and appending.

2) Closing a File

After operations are complete, files should be closed to free resources and ensure data integrity. Use the `fclose()` function.

Example:

fclose(file);

3)Writing to a File

a) Writing Characters – `fputc()`

`Writes a single character to the file.`

`Example:`

```
FILE *file = fopen("example.txt", "w");
fputc('A', file);  // Writes 'A' to the file
fclose(file);
```

b) Writing Strings – fputs()

`Writes a string to the file.`

`Example:`

```
FILE *file = fopen("example.txt", "w");
fputs("Hello, World!", file);
fclose(file);
```

c) Writing Binary Data – fwrite()

Writes blocks of binary data.

Example:

```
int numbers[] = {1, 2, 3, 4};
FILE *file = fopen("data.bin", "wb");
fwrite(numbers, sizeof(int), 4, file);
fclose(file);
```

4) Reading from a File

a) Reading Characters – fgetc()

Reads a single character from the file.

Example:

```
FILE *file = fopen("example.txt", "r");
char c = fgetc(file);  // Reads one character
printf("%c\n", c);
fclose(file);
```

b) Reading Strings – fgets()

Reads a string until a newline or EOF.

Example:

```
FILE *file = fopen("example.txt", "r");
char buffer[50];
fgets(buffer, 50, file);  // Reads up to 49 characters
printf("%s", buffer);
fclose(file);
```

c) Reading Binary Data – fread()

Reads blocks of binary data.

xample:

```
int numbers[4];
FILE *file = fopen("data.bin", "rb");
fread(numbers, sizeof(int), 4, file);
for (int i = 0; i < 4; i++) {
    printf("%d ", numbers[i]);
}
fclose(file);
```

Functions:

Function	Purpose
fopen()	Open a file in a specified mode.
fclose()	Close an open file.
fgetc()	Read a character from the file.
fgets()	Read a string from the file.
fputc()	Write a character to the file.
fputs()	Write a string to the file.
fread()	Read binary data from the file.
fwrite()	Write binary data to the file.