

# Software Engineering Assignment

## MODULE: 2

### Introduction To Programming

#### 1. Overview of C 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. Introduction :**

The C programming language, created in the early 1970s, has been one of the most influential languages in the history of computing.

#### ❖ The History and Evolution of C Programming :

The C programming language was developed in the early 1970s by Dennis Ritchie at Bell Labs, building on earlier languages like B. Initially designed for system programming, C was used to rewrite the Unix operating system, demonstrating its power and portability. The language's efficiency, low-level memory manipulation, and simplicity made it a perfect fit for operating systems, compilers, and embedded systems.

In 1983, the ANSI committee standardized C, which improved consistency across platforms. Later, the ISO adopted this standard in 1990, and updates like C99 and C11 added features such as inline functions and better multi-threading support. C also influenced the creation of other programming languages like C++.

#### ❖ Importance of C :

C remains crucial because of its:

1. **Portability:** C programs can run on different hardware platforms with minimal changes.
2. **Efficiency:** C offers fine-grained control over system resources, making it ideal for performance-critical applications.
3. **Low-Level Access:** C allows direct memory manipulation, essential for system programming and hardware interfacing.
4. **Foundation for Other Languages:** Many modern languages like C++, C#, and Java are based on C's syntax and concepts.

### ❖ Why C is Still Used Today :

Despite the rise of newer languages, C is still widely used because:

1. **Embedded Systems:** C is the dominant language for programming embedded systems and IoT devices due to its efficiency and control over hardware.
2. **Operating Systems:** Most major operating systems, including Linux, are written in C.
3. **Performance:** C is essential for high-performance applications, such as databases, gaming engines, and scientific simulations.
4. **Legacy Systems:** A large amount of existing software is written in C, requiring ongoing maintenance and development.

In conclusion, C's combination of portability, efficiency, low-level access, and influence on modern programming ensures its continued relevance in technology and software development.

---

## 2. Setting Up Environment:

### 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.** To start programming in C, you'll need two main components: a C compiler (such as GCC) and an Integrated Development Environment (IDE) that provides tools to write, compile, and run C programs. Here's a step-by-step guide to help you set up GCC and an IDE, using examples like Dev C++, VS Code, or CodeBlocks.

### Step 1: Installing GCC Compiler For Windows:

#### 1. Download MinGW (Minimalist GNU for Windows):

- Visit the MinGW website or MinGW-w64 (the recommended version).
- Download the MinGW installer or executable for your version of Windows.

#### 2. Install MinGW:

- Run the installer and select the "mingw32-gcc-g++" package (which includes the GCC compiler).
- Follow the installation instructions and choose a folder for the MinGW installation (e.g., C:\MinGW).

### 3. Add MinGW to the System Path:

- Open **Control Panel** → **System** → **Advanced system settings** → **Environment Variables**.
- Under **System variables**, find **Path**, then click **Edit**.
- Add the path to the MinGW **bin** directory (e.g., `C:\MinGW\bin`).
- Click **OK** to save and close all windows.

### 4. Verify the Installation:

- Open **Command Prompt** and type `gcc --version`. If installed correctly, you'll see the version of GCC.

## Step 2: Installing an IDE

### Option 1: DevC++ (Windows)

#### 1. Download DevC++:

- Visit the official DevC++ website: [DevC++ Download](#).
- Download the installer (e.g., `Dev-C++_5.11_setup.exe`).

#### 2. Install DevC++:

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

#### 3. Configure GCC in DevC++:

- When you first launch DevC++, it should automatically detect your GCC compiler if MinGW is installed.
- If needed, go to **Tools** → **Compiler Options** → **Settings**, and check that the paths for GCC are set correctly.

#### 4. Write and Compile Code:

- Open DevC++, create a new project, write your C code, and press **F9** to compile and run your program.

### Option 2: CodeBlocks (Windows, macOS, Linux)

#### 1. Download CodeBlocks:

- Visit the official CodeBlocks website.
- Download the version with MinGW (for Windows) or the appropriate version for your OS.

#### 2. Install CodeBlocks:

- Run the installer and follow the on-screen instructions. For Windows, select the version that comes with MinGW, which includes the GCC compiler.

### 3. Verify Compiler Setup:

- After installation, launch CodeBlocks. It should automatically detect the GCC compiler.
- Go to **Settings** → **Compiler**, and ensure that the paths for GCC are set correctly under the **Toolchain Executables** tab.

### 4. Write and Compile Code:

- Create a new project, write your C code, and hit **F9** to compile and run your program.

## Option 3: Visual Studio Code (VS Code) (Cross-platform)

### 1. Install Visual Studio Code:

- Download VS Code from [Visual Studio Code website](https://code.visualstudio.com/).
- Follow the installation instructions for your operating system.

### 2. Install the C/C++ Extension:

- Launch VS Code and open the **Extensions** view by clicking the Extensions icon on the Activity Bar or by pressing **Ctrl+Shift+X**.
- Search for "C/C++" and install the extension provided by Microsoft.

### 3. Install GCC Compiler (if not already installed):

- On **Windows**, you can install MinGW as explained earlier, or use the Windows Subsystem for Linux (WSL).
- On **macOS** and **Linux**, GCC should already be available or can be installed via the package manager (e.g., `sudo apt install build-essential` for Linux).

### 4. Configure the Build System:

- Create a **tasks.json** file to set up the build system:
  - Go to **Terminal** → **Configure Default Build Task** → Select C/C++: gcc build active file.
  - This creates a `tasks.json` file to automate the process of compiling and running C programs.

### 5. Write and Compile Code:

- Create a new file with `.c` extension, write your C code, and use **Ctrl+Shift+B** to compile and run the program in VS Code.

---

## 3. Basic Structure of a C Program:

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

**Ans.** 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.

❖ Basic Structure of a C Program :

1. **Headers:**

Headers are files that provide declarations for functions and macros used in a program. Common headers include `<stdio.h>` for input/output functions and `<stdlib.h>` for memory allocation and utilities.

2. **Main Function:**

The `main()` function is the entry point of a C program. Execution starts here, and it typically returns an integer (`int`), where `return 0;` indicates successful completion.

3. **Comments:**

Comments are used to explain code and are ignored by the compiler. There are two types:

- **Single-line comments:** `// This is a comment`
- **Multi-line comments:** `/* This is a comment */`

4. **Data Types:**

Data types define the type of data a variable can hold. Common types include:

- `int`: Integer values.
- `float`: Floating-point numbers (decimals).
- `char`: Single characters.
- `double`: Double-precision floating-point numbers.

5. **Variables:**

Variables store data and must be declared with a specific data type. For example, `int x;` declares an integer variable `x`, and `x = 10;` assigns a value to it.

---

❖ Operators in C

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

➤ **Ans.** An Operator is A Symbol that used for perform on specific task .

## ❖ Arithmetic Operators :

These are used to perform mathematical operations on numeric data types.

- `+` : Addition (e.g., `a + b`)
- `-` : Subtraction (e.g., `a - b`)
- `*` : Multiplication (e.g., `a * b`)
- `/` : Division (e.g., `a / b`)
- `%` : Modulus (remainder after division) (e.g., `a % b`)

## ❖ Relational Operators :

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

- `==` : Equal to (e.g., `a == b`)
- `!=` : Not equal to (e.g., `a != b`)
- `>` : Greater than (e.g., `a > b`)
- `<` : Less than (e.g., `a < b`)
- `>=` : Greater than or equal to (e.g., `a >= b`)
- `<=` : Less than or equal to (e.g., `a <= b`)

## ❖ Logical Operators :

These are used to perform logical operations and combine multiple conditions.

- `&&` : Logical AND (true if both operands are true) (e.g., `a && b`)
- `||` : Logical OR (true if at least one operand is true) (e.g., `a || b`)
- `!` : Logical NOT (negates the truth value) (e.g., `!a`)

## ❖ Assignment Operators :

These are used to assign values to variables.

- `=` : Simple assignment (e.g., `a = 5`)
- `+=` : Add and assign (e.g., `a += 5` is equivalent to `a = a + 5`)
- `-=` : Subtract and assign (e.g., `a -= 5`)
- `*=` : Multiply and assign (e.g., `a *= 5`)
- `/=` : Divide and assign (e.g., `a /= 5`)
- `%=` : Modulus and assign (e.g., `a %= 5`)

## ❖ Increment/Decrement Operators :

These are used to increase or decrease the value of a variable by 1.

- `++` : Increment by 1 (e.g., `a++` or `++a`)
- `--` : Decrement by 1 (e.g., `a--` or `--a`)

## ❖ Bitwise Operators :

These perform bit-level operations on integers.

- `&` : Bitwise AND (e.g., `a & b`)
- `|` : Bitwise OR (e.g., `a | b`)
- `^` : Bitwise XOR (e.g., `a ^ b`)
- `~` : Bitwise NOT (e.g., `~a`)
- `<<` : Left shift (e.g., `a << 2`)
- `>>` : Right shift (e.g., `a >> 2`)

## ❖ Conditional (Ternary) Operator :

This is a shorthand for if-else statements. :

- `?:` : Conditional operator (e.g., `condition ? expr1 : expr2`)
  - If `condition` is true, `expr1` is evaluated; otherwise, `expr2` is evaluated.
- 

## ❖ Control Flow Statements in C

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

➤ **Ans.** In C, **decision-making statements** allow the program to make choices based on certain conditions.

- **The main decision-making statements in C are if, else, nested if-else, and switch.**

1. **if Statement :** The if statement evaluates a condition and executes the code inside the block if the condition is true.

**Syntax:**

```
if (condition) {  
    // Block of code  
    // code to execute if condition is true  
}
```

### Example:

```
#include <stdio.h>
int main() {
    int a = 10;
    if (a > 5) {
        printf(" a is greater than 5\n");
    }
}
```

Output:

a is greater than 5

## 2. else Statement :

the else statement is used with if to execute a block of code when the if condition is **false**.

### Syntax:

```
if (condition) {
    // code if condition is true
} else {
    // code if condition is false
}
```

### Example:

```
#include <stdio.h>
int main() {
    int a = 3;
    if (a > 5) {
        printf("a is greater than 5\n");
    } else {
        printf("a is less than or equal to 5\n");
    }
}
```

Output:

a is less than or equal to 5



### 3. if-else if-else (Nested if-else):

The `else if` statement is used when you have multiple conditions to check. If the first condition fails, it checks the next condition in the `else if` block, and so on.

#### Syntax:

```
if (condition1) {  
    // code to execute if condition1 is true  
} else if (condition2) {  
    // code to execute if condition2 is true  
} else {  
    // code to execute if neither condition1 nor condition2 is true  
}
```

#### Example:

```
#include <stdio.h>  
int main() {  
    int num = 7;  
  
    if (num > 10) {  
        printf("The number is greater than 10.\n");  
    } else if (num == 7) {  
        printf("The number is 7.\n");  
    } else {  
        printf("The number is less than or not equal to 7.\n");  
    }  
}
```

#### Output:

The number is 7.

## 4. switch Statement :

The `switch` statement is used when you have multiple values for a single variable and you want to execute different code based on the value of that variable. The `switch` statement evaluates an expression and matches its value to the corresponding `case` label.

**Syntax:**

```
switch (expression) {
    case value1:
        // code to execute if expression == value1
        break;
    case value2:
        // code to execute if expression == value2
        break;
    // more cases...
    default:
        // code to execute if no case matches
}
```

- **break:** Prevents the program from executing the subsequent `case` blocks once a match is found.
- **default:** Executes if no `case` matches the expression.

**Example:**

```
#include <stdio.h>
int main() {
    int day = 3;

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

```
        case 4:
            printf("Thursday\n");
            break;
        case 5:
            printf("Friday\n");
            break;
        case 6:
            printf("Saturday\n");
            break;
        case 7:
            printf("Sunday\n");
            break;
        default:
            printf("Invalid day\n");
    }

    return 0;
}
```

### Output:

Wednesday

//In this example, the value of day is 3, so the program matches case 3 and prints "Wednesday".

### Summary:

- **if:** Executes a block of code if the condition is true.
- **else:** Executes a block of code if the condition is false.
- **else if:** Used to check multiple conditions.
- **switch:** Executes different blocks of code based on the value of a single variable.

---

### ❖ Looping in C :

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

- **While Loops, For Loops, and Do-While Loops** are all control structures used for repeating a block of code based on a condition. However, they differ in their syntax, behavior, and typical use cases. Here's a comparison of each loop type:

### Key Differences

Feature	While Loop	For Loop	Do-While Loop
Condition check	Before loop body (pre-test)	Before loop body (pre-test)	After loop body (post-test)
Guaranteed execution	May never run if condition is false initially	Executes as long as the condition is true	Always runs at least once
Use case	Condition-based iteration (unknown count)	Iterating over fixed ranges or collections	Needs guaranteed one-time execution
Control variables	Must be managed manually (e.g., increment)	Managed within the loop declaration	Typically involves manual condition update
Ideal for	When the number of iterations is not known in advance	When the number of iterations is known	When at least one iteration is required

### ❖ Comparison of Loops in C :

#### While Loop :

- **Syntax :**

```
while (condition) {  
  
    // code  
  
}
```

- **Condition** is checked before executing the loop body.
- Best for **indeterminate loops**, where you don't know how many times to iterate upfront.
- **Example:** Reading user input until it's valid.

#### For Loop :

- **Syntax :**

```
for (initialization; condition; increment) {  
  
    // code  
  
}
```

- **Condition** is checked before each iteration.
- Best when you know the **exact number of iterations** or when iterating over a range/array.
- **Example:** Looping through an array or performing a task a set number of times.

### Do-While Loop :

- **Syntax :**

```
do {  
  
    // code  
  
} while (condition);
```

- **Condition** is checked after the loop body executes, ensuring the loop runs **at least once**.
- Best when you need to execute the loop body **at least once** before checking the condition.
- **Example:** Displaying a menu and prompting the user until a valid choice is made.

### Summary:

- **While:** Use when iterations depend on a condition, and the number of iterations is **unknown**.
  - **For:** Use when the **exact number** of iterations is known in advance.
  - **Do-While:** Use when the loop needs to execute **at least once** regardless of the condition.
- 

### ❖ Arrays in C :

**Q.7 Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.**

➤ **Ans.**

An **array** in C is a collection of variables of the same type, stored in contiguous memory locations. It allows you to store multiple values under a single variable name, with each value accessed using an index.

❖ **Syntax of Array Declaration :**

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

- **Type** : Data type of array elements (e.g., int, float, char).
- **array\_name** : Name of the array.
- **Size** : The number of elements in the array.

❖ **One-Dimensional Array (1D Array):**

A **one-dimensional array** is a simple list of elements where each element can be accessed using a single index.

● **Declaration and Initialization :**

```
int numbers[5]; // Declare an array of 5 integers
// Initializing the array
int numbers[5] = {1, 2, 3, 4, 5};
```

**Example:**

```
#include <stdio.h>
int main() {
int numbers[5] = {10, 20, 30, 40, 50}; // Accessing and printing elements
for (int i = 0; i < 5; i++) {
printf("numbers[%d] = %d\n", i, numbers[i]);
}
}
```

❖ **Multi-Dimensional Arrays :**

A **multi-dimensional array** is an array of arrays, meaning it contains multiple one-dimensional arrays. The most common type is a **two-dimensional array**, which can be thought of as a table or matrix.

❖ **Two-Dimensional Array (2D Array) :**

A two-dimensional array is often used to represent matrices (rows and columns). It has two indices: one for the row and one for the column.

- Declaration and Initialization :

```
int matrix[3][3]; // Declare a 3x3 array (3 rows and 3 columns)

// Initializing a 2D array

int matrix[3][3] = {    {1, 2, 3},
                        {4, 5, 6},
                        {7, 8, 9}
                      };
```

- Example:

```
#include <stdio.h>

int main() {


int matrix[3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };

// Accessing and printing elements of a 2D array

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

}
```

### Key Differences:

Feature	One-Dimensional Array	Multi-Dimensional Array
Structure	A simple list of elements (1D).	An array of arrays (2D or more).
Accessing Elements	<code>array[index]</code> .	<code>array[row][column]</code> for 2D, <code>array[x][y][z]</code> for 3D.
Example	<pre>int numbers[3] = {1, 2, 3};</pre>	<pre>int matrix[2][2] = {{1, 2}, {3, 4}};</pre>
Use Cases	Storing simple lists of data. 	Representing tables, grids, matrices, or complex data.

#### ● Summary:

- **One-Dimensional Array:** A linear list of elements, accessed by a single index.
  - **Multi-Dimensional Array:** An array of arrays, typically used to represent tables or matrices, accessed by multiple indices.
- 

## ❖ Functions in C :

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

- Ans. In C programming, **functions** are blocks of code that perform a specific task. Functions allow us to organize our code into smaller, reusable parts, making it more modular, maintainable, and easier to understand. A function can take some inputs (called parameters), perform some operations, and optionally return a value.

### 1. Function Declaration (Prototype)

- Tells the compiler about the function's name, return type, and parameters.
- **Syntax:**



-By Aakash Prajapati

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

### ● Example:

```
int add(int, int);
```

### ● Function Definition:

- Provides the actual implementation of the function.

### Syntax:

```
return_type function_name(parameter_list) {  
    // function body  
}
```

### Example:

```
int add(int a, int b) {  
    return a + b; // Adds two integers and returns the result  
}
```

### ● Function Call:

- Invokes the function and executes its code.

### Example:

```
int result = add(5, 3); // Calls 'add' with 5 and 3 as arguments
```

### Full Example:

-By Aakash Prajapati

```
#include <stdio.h>

// Function Declaration
int add(int, int);

int main() {
    int x = 5, y = 3;
    int sum = add(x, y);    // Function Call
    printf("Sum: %d\n", sum);
    return 0;
}

// Function Definition

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

### Summary:

**Declaration:** Tells the compiler about the function.

**Definition:** Specifies the function's code.

**Call:** Executes the function.

---

### ❖ Arrays in C

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

➤ **Ans.** An **array** in C is a collection of elements of the same type, stored in contiguous memory locations. It allows you to store multiple values in a single variable, making it easier to manage related data.

## 1. One-Dimensional Arrays :

A one-dimensional array is a linear collection of elements, where each element can be accessed by a single index.

Example:

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

```
main() {
```

```
    // Declaration of a one-dimensional array of integers
```

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

```
    // Accessing array elements
```

```
    printf("First element: %d\n", arr[0]); // Output: 1
```

```
    printf("Third element: %d\n", arr[2]); // Output: 3
```

```
}
```

## 2. Multi-Dimensional Arrays :

A multi-dimensional array is an array of arrays. These arrays can have more than one index to access each element, which allows for organizing data in a tabular or grid-like structure. The most common multi-dimensional arrays are two-dimensional arrays (like matrices), though arrays can have more than two dimensions.

### Example of Multi -Dimensional Array:

```
#include <stdio.h>

int main() {

    // Declaration of a 3D array (2 layers, 3 rows, 4 columns)

    int arr[2][3][4] = {
        {
            {1, 2, 3, 4},
            {5, 6, 7, 8},
            {9, 10, 11, 12}
        },
        {
            {13, 14, 15, 16},
            {17, 18, 19, 20},
            {21, 22, 23, 24}
        }
    };

    // Accessing an element from the 3D array

    printf("Element at layer 1, row 2, column 3: %d\n", arr[0][1][2]); //
Output: 7
    printf("Element at layer 2, row 3, column 4: %d\n", arr[1][2][3]); //
Output: 24

    return 0;
}
```

## Key Differences Between One-Dimensional and Multi-Dimensional Arrays

Aspect	One-Dimensional Array	Multi-Dimensional Array
Structure	A simple list of elements	An array of arrays (table, matrix, etc.)
Accessing Elements	Uses a single index <code>arr[i]</code>	Uses multiple indices <code>arr[i][j]</code> or <code>arr[i][j][k]</code>
Memory Allocation	Contiguous memory for all elements	Elements are arranged in multiple rows, columns, etc.
Use Case	Used for linear data storage	Used for storing data in tables, grids, matrices, etc.
Example	<pre>int arr[5] = {1, 2, 3, 4, 5};</pre>	<pre>int matrix[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};</pre>
Dimensionality	1 dimension	2 or more dimensions (2D, 3D, etc.)

### Conclusion :

- A one-dimensional array stores a list of elements in a linear fashion.
- A multi-dimensional array organizes data in a matrix or table format, with each element requiring multiple indices to access it.
- The choice between using a one-dimensional or multi-dimensional array depends on how the data is organized and accessed.

---

### ❖ Pointers in C :

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

➤ **Ans :**

In C, a pointer is a variable that stores the memory address of another variable. Instead of holding a value directly, a pointer holds the location of where the value is stored in memory. This allows C programs to work directly with memory and perform efficient operations like dynamic memory allocation, array manipulation, and function handling.

● **Why Pointers Are Important in C :**

- Pointers are fundamental in C because they provide several advantages:

1. **Efficiency:** Pointers allow direct manipulation of memory, which can lead to more efficient programs, especially when dealing with large data structures or arrays.

2. **Dynamic Memory Allocation:** Pointers are used for dynamic memory allocation (e.g., using malloc, calloc, realloc, and free), which lets programs allocate memory at runtime.

3. **Function Argument Passing:** Pointers enable call-by-reference argument passing, allowing functions to modify variables in the calling function.

4. **Linked Data Structures:** Pointers are used in implementing data structures like linked lists, trees, graphs, etc.

## ● Pointer Syntax and Declaration :

### 1. Declaring a Pointer :

- To declare a pointer, you use the \* symbol.

**The declaration syntax is as follows:**

**type \*pointerName;**

- type is the data type the pointer will point to (e.g., int, char, float).
- \* indicates that the variable is a pointer (not a regular variable).
- pointer\_name is the name of the pointer variable.

## ● Example of Pointer Declaration :

```
int *ptr; // Declare a pointer to an integer
```

```
char *ch_ptr; // Declare a pointer to a char
```

- To **initialize** a pointer, you assign it the address of a variable using the address-of operator (&).

## ● Example of Pointer Initialization :

```
int num = 10;  
int *ptr = &num; // 'ptr' now points to the address of 'num'
```

## Example Code :

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

    int num = 42;
    int *ptr = &num;          // Pointer 'ptr' stores the address of 'num'

    // %p prints the address stored in ptr
    printf("Address of num: %p\n", (void*)ptr)
    // %p prints the address stored in ptr
    printf("Value of num: %d\n", *ptr); // Dereference ptr to get the value of num

}
```

### ● Output:

Address of num: 0x7ffefee1b45c

Value of num: 42

---

## ❖ Strings in C :

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

➤ Ans.



## 1. strlen()

### Purpose:

- The strlen() function in C is used to determine the length of a null-terminated string (i.e., a string that ends with the null character '\0'&NULL).

### Syntax:

```
size_t strlen (const char *str) ;
```

### Example:

```
#include <stdio.h>

main() {
    const char *str = "Hello, world!";

    printf("Length of the string: %zu\n", strlen(str)); // Output: 13
}
```

## 2. strcpy()

### Purpose:

- The strcpy() function is used to copy the contents of one string into another. It copies the source string into the destination string, including the null terminator.

### Example:

```
#include <stdio.h>
int main() {
    char src[] = "Hello, world!";
    char dest[50];

    strcpy(dest, src); // Copy src to dest
    printf("Copied string: %s\n", dest); // Output: Hello,
world!
}
```

## 3. strcat()

### Purpose:

- The strcat() function is used to concatenate (append) one string to the end of another.

### Example:

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

    char data[50] = "Hello, ";
    char src[] = "world!";
    strcat(data, src); // Concatenate src to dest
    printf("Concatenated string: %s\n", data); // Output: Hello, world!
}
```

## 4. strcmp()

- used for comparision

**example :**

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

    int result = strcmp(str1, str2);
    if (result < 0)
        printf("\'%s\' is less than \\'%s\'\\n", str1, str2);    // Output:
apple is less than banana
    else if (result == 0)
        printf("\'%s\' is equal to \\'%s\'\\n", str1, str2);
    else
        printf("\'%s\' is greater than \\'%s\'\\n", str1, str2);

}
```

## 5. strchr()

- it's used for search character

Example:

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

    if (result != NULL)
        printf("Found 'o' at position: %ld\\n", result - str);    // Output: Found 'o' at
position: 4
    else
        printf("\'o' not found.\\n");

}
```

Summary of Use Cases:

- 1.strlen() is useful when you need to know the length of a string.
  - 2.strcpy() is essential for copying one string to another.
  - 3.strcat() is handy when concatenating strings together.
  - 4.strcmp() is useful for comparing strings lexicographically.
  - 5.strchr() is helpful for finding a specific character in a string.
- 

## ❖ Structures in C :

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

➤ Ans.

### ● Concept of Structures in C :

In C, a **structure** is a user-defined data type that allows you to combine variables of different data types into a single unit. Each variable within a structure is called a **member** (or **field**), and it can have different data types. Structures are useful for representing real-world entities that have multiple attributes. For example, you can represent a **student** with attributes like **name**, **age**, and **grade**.

**Syntax to declare a structure:**

```
struct StructureName {  
    data_type member1;  
    data_type member2;  
    // Other members  
};
```

### Example:

```
struct Person {  
    char name[50]; // String to store the name  
    int age;       // Integer to store the age  
    float height;  // Float to store the height  
};
```

### Initializing :

- You can also declare a structure variable first and then assign values to its members using dot operator (.).

### Example:

```
struct Person {  
    char name[50];  
    int age;  
    float height;  
};  
  
int main() {  
    struct Person person2;  
  
    // Assigning values to the structure members  
    strcpy(person2.name, "Bob"); // Assign string to name  
    person2.age = 30;             // Assign integer to age  
    person2.height = 5.8;         // Assign float to height  
  
    printf("Name: %s\n", person2.name);  
    printf("Age: %d\n", person2.age);  
    printf("Height: %.2f\n", person2.height);  
}
```

## Accessing Structure Members:

- Structure members are accessed using the **dot operator** (.). You use the structure variable's name followed by the member name to access a particular attribute.

### Example:

```
#include <stdio.h>
struct Person {
    char name[50];
    int age;
    float height;
};
int main() {
    struct Person person = {"John", 28, 6.1};

    // Accessing structure members using dot operator
    printf("Name: %s\n", person.name);
    printf("Age: %d\n", person.age);
    printf("Height: %.2f\n", person.height);
}
```

---

## ❖ File Handling in C :

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 is a crucial aspect of programming because it allows programs to interact with external data stored on disk, enabling the persistence of information across program executions. Without file handling, data would be lost once the program terminates. File handling in C enables:

1. **Data Persistence:** Store and retrieve data from files so that information can persist between program runs.
2. **Large Data Handling:** Files provide a way to handle larger datasets that may not fit in memory.
3. **Data Sharing:** Allow different programs or users to access and share data stored in files.
4. **System Communication:** Programs often need to log information or communicate with other systems through files.

## ● File Operations in C :

There are several basic operations that can be performed on files in C: **opening, reading, writing, and closing**. Here's an overview of each operation:

### 1. Opening a File :

Before performing any operations on a file, it must be opened using the **fopen()** function. The **fopen()** function takes two parameters: the name of the file and the mode in which the file should be opened.

-filename: The name of the file you want to open.

mode: The mode in which to open the file. Common modes include:

- **"r"**: Read mode (file must exist).
- **"w"**: Write mode (creates a new file or truncates an existing file).
- **"a"**: Append mode (writes data at the end of the file).
- **"rb", "wb", "ab"**: Binary modes for reading, writing, and appending respectively.
- **"r+"**: Read/Write mode (file must exist).
- **"w+"**: Write/Read mode (creates a new file or truncates an existing file).
- **"a+"**: Append/Read mode (creates a new file if it does not exist).

- **Example:**

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

## 2. Reading from a File :

- To read data from a file, you can use functions such as **fgetc()**, **fgets()**, or **fread()**.

- **fgetc()**: Reads a single character from the file.
- **fgets()**: Reads a line or a set number of characters from the file.
- **fread()**: Reads binary data into a buffer.

- **Example:**

```
char c;
while ((c = fgetc(file)) != EOF) {
    putchar(c);                // Output the character to console
}
```

## 3. Writing to a File :

- To write data to a file, you can use functions like **fputc()**, **fputs()**, or **fwrite()**.

**fputc()**: Writes a single character to the file.

**fputs()**: Writes a string (without a newline) to the file.

**fwrite()**: Writes binary data to the file.



## 5. Closing a File :

- After performing all file operations, you should always close the file using the `fclose()` function to ensure that all buffered data is written to the file and resources are freed.

### Example Code : File Handling in C

```
#include <stdio.h>

int main() {
    FILE *file;
    char buffer[100];

    // Opening the file in read mode

    file = fopen("example.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
    }

    // Reading and displaying contents of the file

    printf("File contents:\n");
    while (fgets(buffer, sizeof(buffer), file)) {
        printf("%s", buffer);
    }

    // Closing the file after reading

    fclose(file);

    // Opening the file in write mode (creates the file or overwrites it)

    file = fopen("example.txt", "w");
    if (file == NULL) {
```

-By Aakash Prajapati

```
        perror("Error opening file for writing");  
  
    }  
  
    // Writing to the file  
    fprintf(file, "Hello, this is a new file content.\n");  
  
    // Closing the file after writing  
    fclose(file);  
  
}
```

### ● Summary of Functions:

- Opening Files: `fopen()`
- Reading Files: `fgetc()`, `fgets()`, `fread()`
- Writing Files: `fputc()`, `fputs()`, `fwrite()`, `fprintf()`
- Closing Files: `fclose()`

-By Aakash Prajapati