Modul-2

---

1. Essay: History and Evolution of C Programming

➔C programming was developed in 1972 by Dennis Ritchie at Bell Labs. It was created to rewrite the Unix operating system and quickly became popular due to its speed and simplicity. C evolved from earlier languages like B and BCPL. Over the years, C has influenced many other programming languages like C++, Java, and C#. It is still widely used today because it is fast, gives low-level access to memory, and is great for system programming, embedded systems, and learning the basics of programming.

---

2. Installing a C Compiler and Setting Up an IDE

➔Steps to install GCC (C compiler):

1. **Go to https://www.mingw-w64.org/ and download the installer.**
2. **Install it and set the system path so you can use gcc in the command line.**

➔Setting up IDEs:

- **DevC++: Download from sourceforge.net, install, and it comes with a built-in compiler.**
- **VS Code: Install from code.visualstudio.com. Add C/C++ extension. Set up tasks to compile using GCC.**
- **CodeBlocks: Download from codeblocks.org with MinGW. Install and start coding.**

---

3. Basic Structure of a C Program

➔#include <stdio.h> // header file

```
int main() { // main function
   // this is a comment
   int a = 10; // variable declaration
   printf("Value of a is %d", a);
   return 0;
}
```

Explanation:

- **#include <stdio.h>: Header file**
- **main(): Starting point of the program**
- **//: Comment line**
- **int: Data type**
- **a: Variable name**

---

4. Types of Operators in C

- **Arithmetic Operators: +, -, *, /, %**
  Example: a + b
- **Relational Operators: ==, !=, >, <, >=, <=**
  Example: a > b
- **Logical Operators: &&, ||, !**
  Example: a > 0 && b > 0
- **Assignment Operators: =, +=, -=, *=, /=, %=**
  Example: a += 5
- **Increment/Decrement: ++, --**
  Example: a++, --b
- **Bitwise Operators: &, |, ^, ~, <<, >>**
  Example: a & b
- **Conditional Operator: ? :**
  Example: a > b ? a : b

---

5. Decision-Making Statements in C

➔**if:**

```
if (a > 0) {
   printf("Positive number");
}
```

if-else:

```
if (a > 0) {
   printf("Positive");
} else {
   printf("Non-positive");
}
```

nested if-else:

```
if (a > 0) {
   if (a < 100) {
      printf("Between 1 and 99");
   }
}
```

switch:

```
int choice = 1;
```

```
switch (choice) {
    case 1: printf("One"); break;
    case 2: printf("Two"); break;
    default: printf("Other");
}
```

---

6. Loops in C

➔while loop:

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

Use when the number of iterations is unknown.

➔for loop:

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

Use when the number of iterations is known.

➔do-while loop:

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

Executes at least once, even if the condition is false.

---

7. break, continue, and goto in C

- **break: Stops the loop.**

➔for (int i = 0; i < 10; i++) {

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

➔**continue: Skips the current iteration.**

```
for (int i = 0; i < 5; i++) {
   if (i == 2) continue;
   printf("%d\n", i);
}
```

➔**goto: Jumps to a label.**

```
int a = 5;
if (a == 5) goto skip;
printf("This won't print");
skip:
printf("Jumped here\n");
```

---

8. Functions in C

• THEORY EXERCISE:

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

  ➔ In C programming, a function is a self-contained block of code designed to perform a specific task. Functions promote code reusability, modularity, and readability.

➔A function declaration, also known as a function prototype, informs the compiler about a function's existence before its actual definition.

- ⇨ **return_type function_name(parameter_type1 parameter_name1, parameter_type2 parameter_name2, ...);**
- ⇨ **int add(int a, int b); // Declares a function named 'add' that takes two integers and returns an integer.**

➔ The function definition provides the actual implementation of the function, containing the code that performs the intended task.

- ⇨ **return_type function_name(parameter_type1 parameter_name1, parameter_type2 parameter_name2, ...) {**
- ⇨ **// Function body: statements to perform the task**
- ⇨ **// return value; (if return_type is not void)**
- ⇨ **}**

➔ To execute the code within a function, you "call" or "invoke" it. When calling a function, you provide actual values (arguments) for its parameters.

- ⇨ **function_name(argument1, argument2, ...);**

➔ #include <stdio.h>

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

int main() {
   int num1 = 10;
   int num2 = 5;
   int result;

   // Function Call
   result = add(num1, num2); // Calls the 'add' function with num1 and num2 as arguments

   printf("The sum is: %d\n", result); // Prints the returned value
   return 0;
}

// Function Definition
int add(int a, int b) {
   int sum = a + b;
   return sum;
}
```

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

```
⇨ #include <stdio.h>
⇨
⇨ // Function Declaration
⇨ long long int calculateFactorial(int n);
⇨
⇨ int main() {
⇨    int number;
⇨    long long int factorialResult;
⇨
⇨    // Prompt user for input
⇨    printf("Enter a non-negative integer: ");
⇨    scanf("%d", &number);
⇨
⇨    // Input validation
```

```
    if (number < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        // Function Call
        factorialResult = calculateFactorial(number);
        printf("The factorial of %d is %lld.\n", number, factorialResult);
    }

    return 0;
}

// Function Definition
long long int calculateFactorial(int n) {
    long long int fact = 1;
    int i;

    // Calculate factorial iteratively
    for (i = 1; i <= n; i++) {
        fact *= i;
    }
    return fact;
}
```

---

9. THEORY EXERCISE:

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

- **In C programming, an array is a collection of elements of the same data type stored in contiguous memory locations. These elements are accessed using a common name and an index, which indicates the position of the element within the array. Array indices in C are zero-based, meaning the first element is at index 0, the second at index 1, and so on.**
- **One-Dimensional Arrays**
- **A one-dimensional (1D) array can be visualized as a linear list or a single row of elements. Each element is accessed using a single index.**
- dataType arrayName[arraySize];
- int numbers[5]; // Declares an integer array named 'numbers' with 5 elements
- numbers[0] = 10; // Assigns 10 to the first element
- numbers[4] = 50; // Assigns 50 to the last element

➔ A multi-dimensional array is an array of arrays, allowing for the storage of data in multiple dimensions, such as rows and columns (like a table or matrix). The most common type is a two-dimensional (2D) array.

- **dataType arrayName[rows][columns];**

⇨ **int matrix[3][4]; // Declares a 2D integer array named 'matrix' with 3 rows and 4 columns**
⇨ **matrix[0][0] = 1; // Assigns 1 to the element at row 0, column 0**
⇨ **matrix[2][3] = 12; // Assigns 12 to the element at row 2, column 3**

➔ Differentiation:

- **Structure:**

  1D arrays are linear lists, while multi-dimensional arrays (e.g., 2D) are structured as grids or tables.

- **Indexing:**

  1D arrays require a single index to access elements, whereas multi-dimensional arrays require multiple indices (one for each dimension).

- **Visualization:**

  1D arrays can be imagined as a single row or column. 2D arrays are like a spreadsheet with rows and columns, and 3D arrays are like stacks of 2D arrays.

➔ LAB EXERCISE:

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

```
⇨ #include <stdio.h>
⇨
⇨ int main() {
⇨     // One-dimensional array
⇨     int oneDArray[5] = {10, 20, 30, 40, 50};
⇨     printf("Elements of one-dimensional array:\\n");
⇨     for (int i = 0; i < 5; i++) {
⇨         printf("%d ", oneDArray[i]);
⇨     }
⇨     printf("\\n\\n");
⇨
⇨     // Two-dimensional array (3x3 matrix)
⇨     int twoDArray[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
⇨     int sum = 0;
⇨
⇨     printf("Elements of two-dimensional array (3x3 matrix):\\n");
⇨     for (int i = 0; i < 3; i++) {
⇨         for (int j = 0; j < 3; j++) {
⇨             printf("%d ", twoDArray[i][j]);
⇨             sum += twoDArray[i][j];
⇨         }
⇨         printf("\\n");
⇨     }
⇨
⇨     printf("\\nSum of all elements in the 3x3 matrix: %d\\n", sum);
```

⇨

⇨ return 0;

⇨ }

---

10. Pointers in C

➔ • THEORY EXERCISE:

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

- ⇨ **In C, pointers are variables that store the memory address of another variable. They provide a powerful way to directly manipulate memory and access data at specific locations, enabling features like dynamic memory allocation and passing data by reference. Pointers are declared using the * symbol, and they are typically initialized with the address of another variable using the & (address-of) operator.**
- ⇨ **data_type *pointer_name;**

➔ Importance of Pointers:

Pointers are crucial in C for several reasons:

1. Memory Manipulation:

Pointers allow direct access and modification of data at specific memory locations, which is essential for tasks like dynamic memory allocation.

2. Dynamic Memory Allocation:

Pointers are used with functions like `malloc()` and `calloc()` to allocate memory dynamically during program execution, enabling flexible memory management.

3. Passing Data by Reference:

Pointers enable passing variables to functions by reference, where the function can modify the original variable's value rather than just a copy, as in pass-by-value.

4. Data Structures:

Pointers are fundamental for implementing complex data structures like linked lists, trees, and graphs, which are commonly used in C.

5. Efficiency:

Pointers can improve efficiency by reducing the need to copy large data structures, especially when passing data between functions.

6. File Handling:

Pointers are used with file pointers (e.g., `FILE*`) to interact with files, allowing you to open, read, and write data.

➔**LAB EXERCISE:**

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

⇨

```
#include <stdio.h>

int main() {
   int myVariable = 10; // Declare and initialize an integer variable
   int *ptr;          // Declare an integer pointer

   printf("Initial value of myVariable: %d\n", myVariable);

   ptr = &myVariable; // Assign the address of myVariable to the pointer ptr

   // Modify the value of myVariable using the pointer
   *ptr = 25;

   printf("Value of myVariable after modification using pointer: %d\n", myVariable);

   return 0;
}
```

---

11. Strings in C

• THEORY EXERCISE:

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

➔**strlen():Find the length of a string excluding '\0' NULL character.**

⇨ **Syntax: strlen(str);**

➔ **strcpy():Copies a string from the source to the destination.**

⇨ **Syntax:strcpy(dest, src);**

➔ **strncpy():Copies n characters from source to the destination.**

⇨ **Syntax:strcpy(dest, src);**

➔ **strcat():Concatenate one string to the end of another.**

⇨ **Syntax: strcat(dest, src);**

➔ **strncat():Concatenate n characters from the string pointed to by src to the end of the string pointed to by dest.**

⇨ **Syntax: strncat(dest, src, n);**

➔ **strcmp():Compares these two strings lexicographically.**

- **Syntax: strcmp(s1, s2);**

➔ strchr():Find the first occurrence of a character in a string. Find the first occurrence of a character in a string.

- **Syntax: strchr(s, c);**

➔ • LAB EXERCISE:

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

```
#include <stdio.h>
#include <string.h> // Required for strcat() and strlen()

int main() {
    char str1[100]; // Declare a character array to store the first string
    char str2[50];  // Declare a character array to store the second string

    printf("Enter the first string: ");
    // Using fgets for safer input, handles spaces and prevents buffer overflow
    fgets(str1, sizeof(str1), stdin);
    // Remove the newline character potentially added by fgets
    str1[strcspn(str1, "\n")] = '\0';

    printf("Enter the second string: ");
    fgets(str2, sizeof(str2), stdin);
    // Remove the newline character potentially added by fgets
    str2[strcspn(str2, "\n")] = '\0';

    // Concatenate str2 to str1. Ensure str1 has enough allocated space.
    strcat(str1, str2);

    printf("\nConcatenated string: %s\n", str1);
    printf("Length of the concatenated string: %zu\n", strlen(str1));

    return 0;
}
```

---

12. Structures in C:

• THEORY EXERCISE: Explain the concept of structures in C. Describe how to declare, initialize, and access structure members.

➔Access Structure Members

To access or modify members of a structure, we use the (.) dot operator. This is applicable when we are using structure variables directly.

➔ **structure_name . member1;**
strcuture_name . member2;

➔ Initialize Structure Members

Structure members cannot be **initialized with the declaration. For example, the following C program fails in the compilation.**

struct ***structure_name {***
data_type1 member1 = value1; // COMPILER ERROR: cannot initialize members here
data_type2 member2 = value2; // COMPILER ERROR: cannot initialize members here
...
};

The reason for the above error is simple. When a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created. So there is no space to store the value assigned.

➔initiation:

By default, structure members are not automatically initialized to 0 or NULL. Uninitialized structure members will contain garbage values. However, when a structure variable is declared with an initializer, all members not explicitly initialized are zero-initialized.

struct structure_name = {0}; // Both x and y are initialized to 0

➔ LAB EXERCISE:

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

```
⇨ #include <stdio.h>
⇨
⇨ // Define the structure for a student
⇨ struct Student {
⇨     char name[50];
⇨     int rollNumber;
⇨     float marks;
⇨ };
```

```
⇨ int main() {
⇨     // Declare an array of 3 Student structures
⇨     struct Student students[3];
⇨
⇨     // Input student details
⇨     for (int i = 0; i < 3; i++) {
⇨         printf("Enter details for student %d:\n", i + 1);
⇨         printf("Name: ");
⇨         scanf("%s", students[i].name);
⇨         printf("Roll Number: ");
⇨         scanf("%d", &students[i].rollNumber);
⇨         printf("Marks: ");
⇨         scanf("%f", &students[i].marks);
⇨     }


⇨     // Print the student details
⇨     printf("\n--- Student Details ---\n");
⇨     for (int i = 0; i < 3; i++) {
⇨         printf("Student %d:\n", i + 1);
⇨         printf("Name: %s\n", students[i].name);
⇨         printf("Roll Number: %d\n", students[i].rollNumber);
⇨         printf("Marks: %.2f\n", students[i].marks);
⇨         printf("----------------------\n");
⇨     }
⇨
⇨     return 0;
⇨ }
```

---

13. File Handling in C

➔ THEORY EXERCISE:

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

File handling in C is crucial because it enables programs to interact with data stored outside their runtime memory, allowing data to persist and be accessed even after the program terminates. This is essential for storing user data, reading configurations, and saving program outputs, among other applications.

File Operations in C:

1. Opening a File:

- `fopen()` is used to open a file, associating it with a program.
- It takes two arguments: the filename and the mode (e.g., "r" for read, "w" for write, "a" for append).
- `fopen()` returns a file pointer, which is used to identify the file in subsequent operations.
- If the file cannot be opened, `fopen()` returns `NULL`.

2. Closing a File:

- `fclose()` is used to close a file, freeing up system resources and ensuring data is properly saved to the disk.
- It takes the file pointer as an argument.
- `fclose()` returns `0` on success and `EOF` on failure.

3. Reading from a File:

- `fgetc()` reads a single character from the file.
- `fgets()` reads a string (up to a specified length or newline) from the file.
- `fscanf()` reads formatted data from the file.
- `fread()` reads a specified number of bytes from the file.

4. Writing to a File:

- `fputc()` writes a single character to the file.
- `fputs()` writes a string to the file.
- `fprintf()` writes formatted data to the file.
- `fwrite()` writes a specified number of bytes to the file.

Example:

➔C

```c
#include <stdio.h>

int main() {
  // Open a file for writing
  FILE *fp = fopen("my_file.txt", "w");
  if (fp == NULL) {
    perror("Error opening file");
    return 1;
  }

  // Write to the file
  fprintf(fp, "This is some text.\n");

  // Close the file
```

```
  fclose(fp);

  // Open the file for reading
  fp = fopen("my_file.txt", "r");
  if (fp == NULL) {
    perror("Error opening file");
    return 1;
  }

  // Read from the file
  char buffer[100];
  while (fgets(buffer, sizeof(buffer), fp) != NULL) {
    printf("%s", buffer);
  }

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

➔ LAB EXERCISE:

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

```
⇨ #include <stdio.h>
⇨ #include <stdlib.h> // Required for exit()
⇨
⇨ int main() {
⇨     FILE *fptr;
⇨     char dataToWrite[] = "This is a test string written to the file.";
⇨     char buffer[100]; // Buffer to store read data
⇨
⇨     // 1. Create and Write to the file
⇨     fptr = fopen("example.txt", "w"); // Open in write mode ("w")
⇨     if (fptr == NULL) {
⇨         printf("Error opening file for writing!\n");
⇨         exit(1); // Exit if file cannot be opened
⇨     }
⇨     fprintf(fptr, "%s", dataToWrite); // Write the string to the file
⇨     fclose(fptr); // Close the file
⇨
⇨     printf("String successfully written to example.txt\n");
⇨
⇨     // 2. Open the file again and Read its contents
⇨     fptr = fopen("example.txt", "r"); // Open in read mode ("r")
⇨     if (fptr == NULL) {
⇨         printf("Error opening file for reading!\n");
```

```
        exit(1); // Exit if file cannot be opened
    }

    printf("\nContents of example.txt:\n");
    // Read and print the contents line by line
    while (fgets(buffer, sizeof(buffer), fptr) != NULL) {
        printf("%s", buffer);
    }

    fclose(fptr); // Close the file after reading

    return 0;
} #include <stdio.h>
#include <stdlib.h> // Required for exit()

int main() {
    FILE *fptr;
    char dataToWrite[] = "This is a test string written to the file.";
    char buffer[100]; // Buffer to store read data

    // 1. Create and Write to the file
    fptr = fopen("example.txt", "w"); // Open in write mode ("w")
    if (fptr == NULL) {
        printf("Error opening file for writing!\n");
        exit(1); // Exit if file cannot be opened
    }
    fprintf(fptr, "%s", dataToWrite); // Write the string to the file
    fclose(fptr); // Close the file

    printf("String successfully written to example.txt\n");

    // 2. Open the file again and Read its contents
    fptr = fopen("example.txt", "r"); // Open in read mode ("r")
    if (fptr == NULL) {
        printf("Error opening file for reading!\n");
        exit(1); // Exit if file cannot be opened
    }

    printf("\nContents of example.txt:\n");
    // Read and print the contents line by line
    while (fgets(buffer, sizeof(buffer), fptr) != NULL) {
        printf("%s", buffer);
    }

```

```
    fclose(fptr); // Close the file after reading

    return 0;
}
```
