**Module - 2 [ Fundamentals of Programing]**

***Introduction to Programming***

1. **Overview of C Programming:**

**THEORY EXERCISE:**

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

**Ans:**

**History and Evolution**

C programming was developed by Dennis Ritchie at Bell Labs in 1972 as an evolution of the B language, primarily for system programming. It gained popularity through its use in developing the Unix operating system. In the 1980s, it was standardized as ANSI C, and over time, updates like C99 and C11 introduced new features. C remains a foundational language, influencing many modern languages and widely used in system-level and performance-critical applications.

**Importance:**

C programming is crucial for system-level programming due to its efficiency, low-level memory access, and portability. It is widely used in developing operating systems, embedded systems, and device drivers. C also forms the foundation for many modern languages like C++, Java, and Python, and is used in performance-critical applications like real-time systems, networking, and gaming. Its role in developing compilers and interpreters further highlights its importance in the software development ecosystem.

**LAB EXERCISE:**

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

**Ans**:

Real-world applications where C programming is widely used in three major areas: operating systems, embedded systems, and game development.

**1. Operating Systems:**

C is the foundation for many operating systems like Unix, Linux, and Windows. Its low-level access to system resources and performance characteristics make it a natural choice for building the core components of an OS, such as the kernel, device drivers, and system utilities.

**2. Embedded Systems:**

Embedded systems, found in devices like microcontrollers, smartphones, and automotive systems, require efficient and resource-conscious programming. C's ability to directly manipulate hardware and its low memory footprint make it suitable for programming these systems, where real-time performance and limited resources are crucial.

**3. Game Development:**

C is heavily used in game development, particularly for game engines and performance-critical components like rendering and physics. Its speed and control over hardware resources are essential for achieving the demanding performance requirements of modern video games.

1. **Setting Up Environment:**

**THEORY EXERCISE:**

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:**

**install a C compiler (e.g., GCC):**

To install a C compiler, first, download and install a compiler like GCCOn Linux, you can install GCC using the command sudo apt-get install gcc. On Windows, you can use MinGW to install GCC. After installation, ensure the compiler is added to the system’s PATH variable to compile C programs from the command line.

**LAB EXERCISE:**

Install a C compiler on your system and configure the IDE. Write your first program to print "Hello, World!" and run it.

**Ans:**

**Writing Your First Program**

To write your first C program, follow these steps:

1. Open your text editor or IDE (like Code::Blocks or Visual Studio Code).

2. Write the following code:

#include

int main() {

printf("Hello, World!\n");

return 0;

}

Save the file with a .c extension (e.g., hello.c).

Compile the program using a C compiler (e.g., gcc hello.c -o hello on the command line).

Run the program (./hello on Linux/Mac or hello.exe on Windows), and it should display "Hello, World!" on the screen.

1. **Basic Structure of a C Program:**

**THEORY EXERCISE:**

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

**Ans**: A C program's basic structure includes preprocessor directives, a main function, comments, data types, and variables. Header files are included using #include directives, which provide access to predefined functions and variables. The main function serves as the entry point, and comments explain the code. Data types specify the kind of data a variable can hold (e.g., int, float, char), and variables store values.

**Examples:**

#include <stdio.h> *// Header file for standard input/output functions*  
  
*// This is a single-line comment*  
*/\* This is a multi-line comment \*/*  
  
int main() { *// The main function, where execution begins*  
 int age = 19; *// Integer variable declaration and initialization*  
 float height = 9.9; *// Float variable declaration and initialization*  
 char grade = 'B'; *// Character variable declaration and initialization*  
  
 printf("Age: %d\n", age); *// Printing the value of age*  
 printf("Height: %.1f\n", height); *// Printing the value of height*  
 printf("Grade: %c\n", grade); *// Printing the value of grade*  
}

**LAB EXERCISE:**

Write a C program that includes variables, constants, and comments. Declare and use different data types (int, char, float) and display their values.

**Ans:**

#include<stdio.h> //Include the standard input/output library

int main() { // This is a single-line comment explaining variable declarations.

int age=9; // Declare and initialize an integer variable

char initial = 'R'; //Declare and initialize a character variable

float price = 19.99; //Declare and initialize a floating-point variable

//Declare a constant using the 'const' keyword

const int MAX\_ATTEMPTS = 3;

/\*

\* This is a multi-line comment.

\* It can span multiple lines to provide more detailed explanations.

\*/

//Display the values of variables

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

printf("Initial: %c\n",initial);

printf("Price: %.2f\n",price);

//Display the value of the constant

printf("Max attempts allowed: %d\n",MAX\_ATTEMPTS);

}

1. **Operators in C:**

**THEORY EXERCISE:**

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

**Ans**: Operators in C operators can be categorized into arithmetic, relational, logical, assignment, increment/decrement, bitwise, and conditional operators.

**Here's a breakdown of each:**

1. Arithmetic Operators:

Used for performing mathematical calculations.

Include:

+ (addition)

- (subtraction)

\* (multiplication)

/ (division)

% (modulo - remainder of a division)

1. Relational Operators:

Used for comparing values.

Return a boolean result (true or false).

Include:

== (equal to)

!= (not equal to)

> (greater than)

< (less than)

>= (greater than or equal to)

<= (less than or equal to)

1. Logical Operators:

Used for combining or modifying boolean expressions.

Return a boolean result.

Include:

&& (logical AND - true if both operands are true)

|| (logical OR - true if at least one operand is true)

! (logical NOT - inverts the boolean value of an operand)

1. Assignment Operators:

Used for assigning values to variables.

The basic assignment operator is =.

Compound assignment operators combine arithmetic operations with assignment (e.g., +=, -=, \*=, /=, %=).

1. Increment/Decrement Operators:

Used to increase or decrease the value of a variable by 1.

++ (increment)

-- (decrement)

Can be used in prefix (e.g., ++x) or postfix (e.g., x++) forms, with subtle differences in behavior.

1. Bitwise Operators:

Used for performing operations on individual bits of data.

Include:

& (bitwise AND)

| (bitwise OR)

^ (bitwise XOR)

~ (bitwise NOT)

<< (left shift)

>> (right shift)

1. Conditional Operator (Ternary Operator):

A shorthand for an if-else statement.

Syntax: condition ? expression1 : expression2

If condition is true, expression1 is evaluated; otherwise, expression2 is evaluated.

**LAB EXERCISE:**

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

**Ans:**

#include<stdio.h>

int main()

{

int num1, num2;

printf("Enter Two integer");

scanf("%d %d",&num1,&num2);

//Arithmetic operations

printf("Arithmetic Operations:\n");

printf("%d + %d = %d\n", num1, num2, num1 + num2);

printf("%d - %d = %d\n", num1, num2, num1 - num2);

printf("%d \* %d = %d\n", num1, num2, num1 \* num2);

if (num2 != 0) {

printf("%d / %d = %d\n", num1, num2, num1 / num2);

printf("%d %% %d = %d\n", num1, num2, num1 % num2);

} else {

printf("Division by zero is not allowed.\n");

}

//Relational operations

printf("\nRelational Operations:\n");

printf("%d == %d is %d\n", num1, num2, num1 == num2);

printf("%d != %d is %d\n", num1, num2, num1 != num2);

printf("%d > %d is %d\n", num1, num2, num1 > num2);

printf("%d < %d is %d\n", num1, num2, num1 < num2);

printf("%d >= %d is %d\n", num1, num2, num1 >= num2);

printf("%d <= %d is %d\n", num1, num2, num1 <= num2);

//Logical operations (assuming int for boolean representation)

printf("\nLogical Operations:\n");

printf("(%d && %d) = %d\n", num1, num2, num1 && num2); // Logical AND

printf("(%d || %d) = %d\n", num1, num2, num1 || num2); // Logical OR

printf("!(%d) = %d\n", num1, !num1); // Logical NOT

printf("!(%d) = %d\n", num2, !num2); // Logical NOT

}

1. **Control Flow Statements in C:**

**THEORY EXERCISE:**

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

**Ans:**

**If Statements**

if statement is the basic decision making statement

Used to decide whether a certain statement or block of statements will be executed or not

If Statements

**Syntax :**

if( condition )

{

statement\_1 ; // true block

statements

}

statement x ;

**Example:**

#include <stdio.h>  
int main() {  
 int age = 90;  
 if (age >= 18) {  
 printf("You are eligible to vote.\n");  
 }  
}

**If else Statements**

if else statement allows selecting any one of the two available options depending upon the output of the test condition.

if else statement

**Syntax :**

if (condition )

{

statements; // true statement

}

else

{

statements ; // false statement

}

**Example:**

#include <stdio.h>  
int main() {  
 int num = 9;  
 if (num % 2 == 0) {  
 printf("%d is even.\n", num);  
 } else {  
 printf("%d is odd.\n", num);  
 }  
}

**Nested if statement**

Nested if statement is simply an if statement embedded with an another if statement.

**Syntax :**

if (condition1)

{

statements ; // executes when condition1 is true

if ( condition2)

{

statements ; // executes when condition2 is true

}

}

**Example:**

#include <stdio.h>  
int main() {  
 int score = 89;  
 char grade;  
  
 if (score >= 90) {  
 grade = 'A';  
 } else {  
 if (score >= 80) { *// Nested if*  
 grade = 'B';  
 } else { *// Nested else*  
 grade = 'C';  
 }  
 }  
 printf("Your grade is: %c\n", grade);  
}

**Switch statements**

Switch case statements are a substitute for long if statements that compare a variable to several integer values.

**Syntax :**

switch ( n) // executed when n = 1

{

case 1 : // executed when n = 2

break;

case 2 : // executed when n doesn’t match any case

break ;

default :

}

**Example:**

#include <stdio.h>  
int main() {  
 char choice = 'B';  
  
 switch (choice) {  
 case 'A':  
 printf("Excellent!\n");  
 break;  
 case 'B':  
 printf("Good!\n");  
 break;  
 case 'C':  
 printf("Fair.\n");  
 break;  
 default:  
 printf("Invalid choice.\n");  
 break;  
 }  
}

**LAB EXERCISE:**

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

**Ans:**

#include<stdio.h>

int main()

{

int number;

int monthNumber;

// Part 1: Check if a number is even or odd using if-else

printf("Enter an integer to check if it's even or odd: ");

scanf("%d", &number);

if (number % 2 == 0) {

printf("%d is an even number.\n", number);

} else {

printf("%d is an odd number.\n", number);

}

//Part 2: Display month name using switch statement

printf("\nEnter a number (1-12) to display the corresponding month name: ");

scanf("%d", &monthNumber);

switch (monthNumber) {

case 1:

printf("January\n");

break;

case 2:

printf("February\n");

break;

case 3:

printf("March\n");

break;

case 4:

printf("April\n");

break;

case 5:

printf("May\n");

break;

case 6:

printf("June\n");

break;

case 7:

printf("July\n");

break;

case 8:

printf("August\n");

break;

case 9:

printf("September\n");

break;

case 10:

printf("October\n");

break;

case 11:

printf("November\n");

break;

case 12:

printf("December\n");

break;

default:

printf("Invalid month number. Please enter a number between 1 and 12.\n");

break;

}

}

1. **Looping in C:**

**THEORY EXERCISE:**

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

**Ans:**

For loops are best when the number of iterations is known beforehand, while loops are suitable when the number of iterations is unknown and depends on a condition, and do-while loops are used when at least one iteration is required regardless of the condition.

**For Loop:**

**Purpose:** Executes a block of code a predetermined number of times.

**Syntax:** for (initialization; condition; increment/decrement) { // code to be executed }

**Use Cases:**

Iterating through arrays or collections where the size is known.

Repeating a task a specific number of times, like counting or processing a fixed set of data.

When the loop control variable (e.g., an index) is needed within the loop.

**While Loop:**

**Purpose:** Repeats a block of code as long as a specified condition is true.

**Syntax:** while (condition) { // code to be executed }

**Use Cases:**

When the number of iterations is not known in advance, like reading data from a file until the end is reached or waiting for a user input.

When the loop needs to continue based on a complex condition that might change during execution.

Often used when dealing with user input or external factors.

**Do-While Loop:**

**Purpose:** Executes a block of code at least once, and then repeats as long as a specified condition is true.

**Syntax:** do { // code to be executed } while (condition);

**Use Cases:**

When you need to ensure that a block of code runs at least one time, regardless of the condition.

Useful in situations where some initialization is needed before the condition can be evaluated.

Examples include menus where the user must see options at least once, or input validation where an initial prompt is required.

**LAB EXERCISE:**

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

**Ans:**

#include<stdio.h>

int main()

{

int i;

// Using for While loop

printf("Number from 1 to 10 using while loop: \n");

i=1; // initialize loop control variable

while(i<=10){

printf("%d \n",i);

i++; // increment

}

printf("\n"); // Add new line

// Using for for loop

printf("Number from 1 to 10 using for loop: \n");

for(i=1; i<=10; i++);{

printf("%d \n",i);

}

printf("\n"); //Add new line

// Using for do-while loop

printf("Number from 1 to 10 using for do-while loop: \n");

i=1; // initialize loop control variable

do{

printf("%d\n",i);

i++;

}

while(i<=10);

printf("\n"); //Add new line

}

1. **Loop Control Statements:**

**THEORY EXERCISE:**

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

**Ans: The Break Statement:**

The break statement is used inside loop or switch statement.

When compiler finds the break statement inside a loop, compiler will abort the loop and continue to execute statements followed by loop.

**Syntax:** break ;

**Example:**

#include <stdio.h>  
 int main() {  
 for (int i = 0; i < 10; i++) {  
 if (i == 5) {  
 break  
 }  
 printf("%d ", i);  
 }  
 printf("\nLoop finished.\n");   
 }

**Continue statement:**

The continue statement is also used inside loop.

When compiler finds the continue statement inside a loop, compiler will skip all the following statements in the loop and resume the next loop iteration.

**Syntax**: continue ;

**Example:**

#include <stdio.t>  
 int main() {  
 for (int i = 0; i < 10; i++) {  
 if (i % 2 == 0) {  
 continue;   
 }  
 printf("%d ", i);  
 }  
 printf("\nLoop finished.\n");  
 }

**The GOTO statement:**

By using this goto statements we can transfer the control from current location to anywhere in the program.

To do all this we have to specify a label with goto and the control will transfer to the location where the label is specified.

**Example:**

#include <stdio.h>  
 int main() {  
 int i = 0;  
   
 loop\_start: *// Label definition*  
 if (i < 5) {  
 printf("%d ", i);  
 i++;  
 goto loop\_start; *// Jump back to loop\_start*  
 }  
   
 printf("\nLoop finished using goto.\n");  
 }

**LAB EXERCISE:**

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

**Ans:**

#include<stdio.h>

int main() {

int i;

for (i = 1; i <= 10; i++) {

// Use continue to skip printing the number 3

if (i == 3) {

continue; // Skips the rest of the current iteration and moves to the next

}

// Use break to stop the loop when the number reaches 5

if (i == 10) {

break; // Exits the loop entirely

}

printf("Number: %d\n", i);

}

printf("Loop terminated.\n");

}

1. **Functions in C:**

**THEORY EXERCISE:**

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

**Ans**: In C programming, a function is a self-contained block of code designed to perform a specific task. Functions promote code reusability, modularity, and organization within a program.

**Function Declaration:**

A function declaration, also known as a function prototype, informs the compiler about a function's name, return type, and the types and order of its parameters. It does not contain the actual implementation of the function. This declaration is typically placed before the main function or in a header file.

**Syntax:**

return\_type function\_name (parameter\_type1 parameter\_name1, parameter\_type2 parameter\_name2, ...);

**Example:**

int add(int a, int b); // Declares a function named 'add' that takes two integers and returns an integer.

**Function Definition:**

The function definition provides the actual implementation of the function's logic.

**Example:**

int add(int a, int b) {  
 int sum = a + b;  
 return sum; *// Returns the sum of a and b*  
}

**Calling a Function:**

To execute the code within a function, it must be called from another part of the program, typically from main or another function. When a function is called, control transfers to the called function, and upon completion, control returns to the point where it was called.

**Example:**

#include <stdio.h>  
  
*// Function Declaration*  
int add(int a, int b);  
  
int main() {  
 int num1 = 10;  
 int num2 = 5;  
 int result;  
  
 *// Calling the 'add' function*  
 result = add(num1, num2);   
  
 printf("The sum is: %d\n", result);   
  
 return 0;  
}  
  
*// Function Definition*  
int add(int a, int b) {  
 int sum = a + b;  
 return sum;  
}

**LAB EXERCISE:**

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

**Ans:**

#include<stdio.h>

// Function Declaration

long long int calculateFactorial(int n);

int main(){

int number;

long long int factorialResult;

// Prompt the user to enter a number

printf("Enter a non-negative integer: ");

scanf("%d", &number);

// Check for negative input

if (number < 0) {

printf("Factorial is not defined for negative numbers.\n");

} else {

// Function Call

factorialResult = calculateFactorial(number);

printf("Factorial of %d is %lld\n", number, factorialResult);

}

}

// Function Definition

long long int calculateFactorial(int n) {

long long int factorial = 1;

// Calculate factorial using a loop

for (int i = 1; i <= n; i++) {

factorial \*= i;

}

return factorial;

}

1. **Arrays in C:**

**THEORY EXERCISE:**

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

**Ans**: In C, an array is a collection of elements of the same data type, stored in contiguous memory locations. It allows you to store multiple values under a single variable name, using an index to access individual elements. One-dimensional (1D) arrays represent a single row or column of data, while multidimensional arrays, like 2D arrays (representing tables or matrices), can store data in multiple dimensions.

**One-Dimensional Arrays:**

**Concept:** A 1D array is a linear sequence of elements, accessed using a single index.

**Declaration:** data\_type array\_name[size];

data\_type specifies the type of elements the array will hold (e.g., int, float, char).

array\_name is the name you give to the array.

size indicates the number of elements the array can store.

**Example:**

int numbers[5] = {10, 20, 30, 40, 50}; *// Array of 5 integers*  
 // numbers[0] is 10, numbers[1] is 20, and so on.

**Multidimensional Arrays:**

**Two-Dimensional Arrays (2D):**

Represented as rows and columns. Think of a spreadsheet or a matrix.

**Declaration:** data\_type array\_name[rows][columns];

**Example (2D):**

int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };  
 // matrix[0][0] is 1, matrix[0][1] is 2, matrix[1][2] is 6

**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.

**Ans:**

#include<stdio.h>

int main(){

// One-dimensional array

int one\_dimensional\_array[5];

printf("Enter 5 integers for the one-dimensional array:\n");

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

scanf("%d", &one\_dimensional\_array[i]);

}

printf("One-dimensional array elements: ");

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

printf("%d ", one\_dimensional\_array[i]);

}

printf("\n");

// Two-dimensional array (3x3 matrix)

int two\_dimensional\_array[3][3];

printf("Enter 9 integers for the 3x3 matrix:\n");

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

for (int j = 0; j < 3; j++) {

scanf("%d", &two\_dimensional\_array[i][j]);

}

}

int sum = 0;

printf("Two-dimensional array (3x3 matrix):\n");

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

for (int j = 0; j < 3; j++) {

printf("%d ", two\_dimensional\_array[i][j]);

sum += two\_dimensional\_array[i][j];

}

printf("\n");

}

printf("Sum of all elements in the matrix: %d\n", sum);

}

1. **Pointers in C:**

**THEORY EXERCISE:**

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

**Ans:**

**Pointers in C :**

Pointers in C are variables that store the memory address of another variable. They are declared using the `\*` symbol, and you can access the value stored at the memory address using the dereference operator `\*`. Pointers allow direct memory manipulation, making them powerful for dynamic memory allocation and working with arrays, functions, and structures. The address of a variable is obtained using the address-of operator `&`.

**Pointer Declaration and Initialization:**

In C, pointers are declared using the \* symbol, and they must be initialized to a valid memory address before use.

**Pointer Declaration:**

int \*ptr; // Declares a pointer to an integer

**Pointer Initialization:**

You can initialize a pointer by assigning it the address of a variable using the address-of operator &:

int x = 10;

int \*ptr = &x; // Pointer ptr holds the address of variable x

Now, ptr points to the memory location where x is stored. You can access the value of x through \*ptr.

**LAB EXERCISE:**

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

**Ans:**

#include<stdio.h>

int main(){

int myVariable = 10; // Declare and initialize an integer variable

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

printf("Original 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 = 20;

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

}

1. **Strings in C:**

**THEORY EXERCISE:**

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

**Ans**: String handling functions in C are essential for manipulating character arrays (strings). strlen() determines string length, strcpy() copies strings, strcat() concatenates them, strcmp() compares them, and strchr() locates a character.

1. strlen():

**Purpose:** Calculates the number of characters in a string, excluding the null terminator ('\0').

**Example:**

#include <stdio.h>  
 #include <string.h>  
  
 int main() {  
 char myString[] = "Hello";  
 size\_t length = strlen(myString);  
 printf("Length of string: %d\n", length); *// Output: Length of string: 5*  
 }

1. strcpy():

**Purpose:** Copies one string to another.

Example:

#include <stdio.h>  
 #include <string.h>  
  
 int main() {  
 char source[] = "Copy this";  
 char destination[20];  
 strcpy(destination, source);  
 printf("Destination string: %s\n", destination);

}

1. strcat():

**Purpose:** Appends one string to the end of another.

Example:

#include <stdio.h>  
 #include <string.h>  
  
 int main() {  
 char str1[20] = "Hello ";  
 char str2[] = "World!";  
 strcat(str1, str2);  
 printf("Concatenated string: %s\n", str1);  
 }

1. strcmp():

**Purpose:** Compares two strings lexicographically (based on their ASCII values).

Example:

#include <stdio.h>  
 #include <string.h>  
  
 int 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("String 1 is less than String 2\n

} else {  
 printf("String 1 is greater than String 2\n");  
 }

}

1. strchr():

**Purpose:** Finds the first occurrence of a character within a string.

Example:

#include <stdio.h>  
 #include <string.h>  
  
 int main() {  
 char str[] = "This is a string";  
 char \*ptr = strchr(str, 's');  
  
 if (ptr != NULL) {  
 printf("First occurrence of 's' found at: %s\n", ptr - str);

} else {  
 printf("Character not found\n");  
 }  
 return 0;  
 }

**LAB EXERCISE:**

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

**Ans:**

#include<stdio.h>

#include<string.h>

int main(){

char str1[100];

char str2[50];

printf("Enter the first string: ");

fgets(str1, sizeof(str1), stdin);

str1[strcspn(str1, "\n")] = 0;

printf("Enter the second string: ");

fgets(str2, sizeof(str2), stdin);

str2[strcspn(str2, "\n")] = 0;

strcat(str1, str2);

printf("Concatenated string: %s\n", str1);

int length = strlen(str1);

printf("Length of the concatenated string: %d\n", length);

}

1. **Structures in C:**

**THEORY EXERCISE:**

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

**Ans:**

Structures in C are user-defined data types that allow the grouping of variables of different data types under a single name.

**Declaration of Structures:**

Structures are declared using the struct keyword, followed by a user-defined name (the tag) and a block containing the declarations of its members. Each member has a data type and a name.

**Example:**

struct Student {  
 char name[90];  
 int roll\_number;  
 float marks;  
};

**Initialization of Structure Members:**

Structure members can be initialized during declaration or after declaration using the dot operator. During declaration (using an initializer list).

**Examlpe:**

struct Student student1 = {"Alice", 119, 85.5};

**After declaration (using the dot operator):**

struct Student student2;  
 strcpy(student2.name, "Bob");   
 student2.roll\_number = 102;  
 student2.marks = 99.0;

**Accessing Structure Members:**

Individual members of a structure variable are accessed using the dot operator (.).

Example:

printf("Student Name: %s\n", student1.name);  
printf("Roll Number: %d\n", student1.roll\_number);  
printf("Marks: %.2f\n", student1.marks);  
  
student2.marks = 93.0; // Modifying a member

**LAB EXERCISE:**

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

**Ans:**

#include<stdio.h>

#include<string.h>

// Define the structure for student details

struct Student {

char name[50];

int roll\_number;

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: ");

fgets(students[i].name, sizeof(students[i].name), stdin);

students[i].name[strcspn(students[i].name, "\n")] = 0;

printf("Roll Number: ");

scanf("%d", &students[i].roll\_number);

printf("Marks: ");

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

getchar();

}

// Print student details

printf("\nStudent 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].roll\_number);

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

}

}

1. **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.

**Ans:**

**Importance File handling in C is done using standard functions:**

* fopen(filename, mode): Opens a file in specified mode ("r", "w", "a", etc.).
* fclose(fp): Closes an opened file.
* fprintf(fp, format, args), fputs(str, fp), fputc(ch, fp): Write data to a file.
* fscanf(fp, format, args), fgets(str, size, fp), fgetc(fp): Read data from a file.

**File Operations (Opening, Closing, Reading, and Writing):**

1. **Opening a File (fopen())**

A file is opened using fopen(filename, mode), where mode can be:

"r" (read), "w" (write), "a" (append)

"r+" (read/write), "w+" (write/read), "a+" (append/read)

FILE \*file = fopen("data.txt", "w"); // Opens file in write mode

if (file == NULL) {

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

return 1;

}

1. **Closing a File (fclose())**

Always close the file using fclose(file); to free resources.

1. **Reading from a File (fscanf() / fgetc() / fgets())**

FILE \*file = fopen("data.txt", "r");

char str[100];

fgets(str, 100, file); // Read a line from file

printf("Read: %s", str);

fclose(file);

1. **Writing to a File (fprintf() / fputc() / fputs())**

fprintf(file, "Hello, File Handling!"); // Write text to file

fclose(file); // Close file after use

**LAB EXERCISE:**

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

**Ans:**

#include<stdio.h>

#include<stdlib.h>

int main() {

FILE \*fp;  
 const char \*data\_to\_write = "This is a test string written to the file.";  
 char buffer[100]; *// Buffer to store read data*  
  
 *// 1. Create and write to the file*  
 fp = fopen("example.txt", "w"); *// Open in write mode ('w')*  
 if (fp == NULL) {  
 perror("Error opening file for writing");  
 return EXIT\_FAILURE;  
 }  
  
 fprintf(fp, "%s\n", data\_to\_write); *// Write the string to the file*  
 printf("Successfully wrote to 'example.txt'\n");  
  
 fclose(fp); *// Close the file*  
  
 *// 2. Open the file again for reading*  
 fp = fopen("example.txt", "r"); *// Open in read mode ('r')*  
 if (fp == NULL) {  
 perror("Error opening file for reading");  
 return EXIT\_FAILURE;  
 }  
  
 printf("\nReading content from 'example.txt':\n");  
 *// Read and display content line by line*  
 while (fgets(buffer, sizeof(buffer), fp) != NULL) {  
 printf("%s", buffer);  
 }  
  
 fclose(fp); *// Close the file*

}