Module 2 – Introduction to Programming

Overview of C Programming

 THEORY EXERCISE:

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

Ans>

C programming is a high level, general-purpose programming language developed in the 1970s. It is widely used for system and application software development due to its efficiency and portability. C Provide low level access and allows to manipulation of hardware, making it ideal for developing operating system, and other performance critical applications.

HISTORY AND EVOLUTION

C programming was developed by Dennis Ritchie at Bell labs in 1972 as an evolution of the b language, primely for system programming. It gain popularity through its use in developing the Unix operating system. In the 1980s, it was standardized as ANSI C, and over time, update like c99 and c11 introduced new feature.

 LAB EXERCISE:

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

2. Setting Up Environment

 THEORY EXERCISE:

o 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

Installing C Complier

To install a C compiler, first download and install a compiler like GCC on 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:

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

Ans

"hello world" program in C

```
#include<stdio.h>

    int main(){
       printf("hello world");
    }
```

3. Basic Structure of a C Program

 THEORY EXERCISE:

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

Ans

C program structure

C program typically consists of three main part:

1. Preprocessor Directives: Instruction like #include to include libraries, which are processed before copilation.
2. Main Function: The entry point of the program, written as int main(), where execution starts.
3. Functions and Statements: The body of program containing logic and functions to perform specific tasks, with the main function returning an intger value (return 0;)

Comments in C language

1.Single line comments: Use // to comment out a single line.

2.multi line comments: Use /* for start and */ for end of comment

Data types : Data type specifies what type of data can store a variable. Data types in C:

1. Basic 2. Derived 3. Enumeration and 4. Void.

Variables: are names gives you in the memory to locating program.
Basic variables: int, float, double, char and void

Constants: Constants are the fixed value in program. Which means that we cannot change it's value.
Ex: const int res = 5;

Types of constants:
Integer, floating, string and character.

Keywords in C:

auto	double	int	struct
Break	else	long	switch
Case	enum	register	typedef
Char	extern	return	union
Const	float	short	unsigned
Continue	for	signed	void
Default	goto	sizeof	volatile
Do	if	static	while

Identifiers :
Identifiers is a name used to identify variable, function, or any user defined item. Ex num1,getchar(),sum, ab_c etc.

 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
//writing program

```
#include<stdio.h>
        /*this is a
                Ex
                        Program*/
        int main(){
              const num = 56;
                int a = 5;
                float b = 5.0;
                char c = a;
                printf("%d and %d and %f and %c",num,a,b,c);

        }
```

4. 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:

1 Arithmetic operators >> +, -, *, /, %, ++, --;
2 Relational operators >> ==, >, <, !(not)=, >=, <=;
3 Logical operators >> A. && logical and true when all condition are true.
B. || logical or and true when one condition is true.
C. ! not and true when condition is false.
4 Assignment operators >> =, -=, +=, *=, /= ;

 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

```
/*
Take input from user
Show operators task
*/
/*<<------source code------>>*/
#include<stdio.h>
#include<stdbool.h>

int main(){
    // Declare the variables:
    int a,b;
    int sum, sub, multi, div;
    int big, small;

    // Take inputs from user:
    printf("Enter a: ");
    scanf("%d",&a);
    printf("Enter b: ");
    scanf("%d",&b);

    // Write the program:
    // 1 arithimatic
    sum = a+b;
    sub = a-b;
    multi = a*b;
    div = a/b;
    printf("Use of Arithimetic Operators :\nSum %d\nSub %d\nMulti %d\nDiv %d\n",sum,sub,multi,div);

    // 2 relational
    if(a>b){
        big = a;
        printf("a is bigger: %d\n",big);
    }else{
        big = b;
        printf("b is bigger: %d\n",big);
```

```
    }
    if(a<b){
        small = a;
        printf("a is smallar: %d\n",small);
    }else{
        small = b;
        printf("b is smallar: %d\n",small);
    }
    if(a==b){
        printf("a & b are equal:\n");
    }else if(a!=b){
        printf("a & b aren't equal:\n");
    }

    // Logical
    int c;
    printf("Enter a value: ");
    scanf("%d",&c);
    if(c>a&&c>b)
    {
        printf("C is Greater");
    }else{
        return 1;
    }



}
```

5. Control Flow Statements in C

 THEORY EXERCISE:

- Explain decision-making statements in C (if, else, nested if-else, switch).

Ans

Control Structure:

Real life situations where we have to condition based decisions by asking ‘if’ questions.

Ex :-

If age is above 18, I am allowed to driving vehicles or else not allowed.

Following types of statements:

1. If:- if statement is the basic decision making statement. Used to decide weather a certain statement will be executed or not.

syntax:

```
if(condition){
    statement_1; //true block
    statements
```

}
Statement x;

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

```
if(condition){
    statement; //true
}else{
    statements; //false
}
```

3 nested if :- nested if statement is simply an if statement embedded with another if statement.
Syntax:

```
if(condition 1){
    statement; executed when condition 1 is true.
    if(condition 2){
        statement; //executed when condition 2 is true
    }
}
```

4 switch:- switch case statement are a substitute for long if statements that compare a variable to several integer values.
syntax:

```
switch(n){
case 1:  //executed when n = 1
    break;
case 2: //executed when n = 2
    break;
default : //executed when n doesn't match any case
    break;
}
```

Nested switch:- nested switch statement occurs when switch statement is defined inside another switch statement.
Syntax:

```
switch(ch1)
{
case 'A': printf ("\n This A is part of outer switch " );
        switch (ch2)
        {
        case 'A ': printf("\n This A is part of inner switch "); break;
        case' B' :
        }
        break ;
        case' B' :
                }
```

```
}
```

Provide examples of each.

 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.).

EVEN ODD PROGRAM USING IF-ELSE STATEMENT:

```
#include <stdio.h>

int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);

    if (num % 2 == 0) {
        printf("%d is even.\n", num);
    } else {
        printf("%d is odd.\n", num);
    }

    return 0;
}
```

SWITCH PROGRAM FOR MONTHS

```
#include<stdio.h>

int main(){
    //take input from user
    int a;
    printf("Enter the month number (1-12): ");
    scanf("%d", &a);
    // program
    switch(a){
        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.\n");
    }
}
```

6. 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

LOOPS IN C

A LOOP STATEMENT ALLOWS US TO EXECUTE A STATEMENT OR GROUP OF STATEMENTS MULTIPLE TIMES BASED ON A CONDITION.

EX:- PRINT 1 TO 100 NUMBERS.

TYPES OF LOOPS

1. **ENTRY CONTROL:** A condition is checked before executing the loop. It also called as a pre-checking loop.
2. **EXIT CONTROL:** A condition is checked after executing the loop. It is called as a post checking loop.

ENTRY CONTROLLED LOOPS

1. **FOR LOOPS:** It is a repetition control structure that allows you to efficiently write loop that needs to execute a specific number of times.
2. **WHILE LOOPS:** It repeatedly executes a target statement as long as the condition is true.

EXIT CONTROLLED LOOPS:

1. **DO WHILE LOOPS:** Do while loop is similar to while loop , except the fact that it execute once even while condition is false.

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

Ans

PRINTING 1 TO 10 USING ALL THREE LOOPS:

```
#include <stdio.h>

int main() {
    //Using for loop
    for (int i = 1; i <= 10; i++) {
        printf("%d\n", i);
    }
    return 0;

    //Using while loop
    int i = 1;
    while (i <= 10) {
        printf("%d\n", i);
        i++;
    }

    //Using do while loop
    int j = 1;
    do {
        printf("%d\n", j);
        j++;
    } while (j <= 10);

}
```

7. Loop Control Statements

 THEORY EXERCISE:
- Explain the use of `break`, `continue`, and `goto` statements in C. Provide examples of each.

Ans:

USE OF BREAK, CONTINUE AND GOTO:

1. **GOTO STATEMENT:** by using 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 other location where the label is specified.
2. **BREAK STATEMENT:** the break statement is used inside loop or switch statement.
   When compiler find the break statement inside a loop, compiler will abort the loop and continue to execute statement followed by loop.
3. **CONTINUE:** the continue statement is also used inside loop.
   When compiler finds the continue statement inside the loop , compiler will skip all the following statements in the loop and resume the next loop iteration.

 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() {
    // Break statement example
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            break;  // Exit the loop when i is 5
        }
        printf("%d\n", i);
    }

    // Continue statement example
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            continue;  // Skip the rest of the loop when i is 5
        }
        printf("%d\n", i);
    }

    return 0;
}
```

8. Functions in C

 THEORY EXERCISE:

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

Ans:

FUNCTION IN C:

A function in C is a named block of code that performs a specific task, which can be reused throughout your program. It helps make your code modular, easier to read, and maintainable

Functions come in two main types:

- **Library functions — built into C (e.g., printf(), scanf()).**
- **User-defined functions — written by the programmer to perform custom tasks.**

Function Definition

- **Contains the actual implementation, including the body of the function.**
- **It matches the declaration's signature**

Example:

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

Call a Function

You call a function using its name, followed by parentheses with arguments (if any). The program jumps to the function's definition, executes the code, and then returns—if there's a return value.

Example:

```
int result = add(5, 3);  // Calls add(), returns 8
```

 LAB EXERCISE:
- Write a C program that calculates the factorial of a number using a function.

Include function declaration, definition, and call.
Ans
MAKING AUR CALLING FUNCTION:

```
#include<stdio.h>
//making function to calculate factorial
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    //taking input from user
    int num;
    printf("Enter a positive integer: ");
    scanf("%d", &num);
    //calling function
    printf("Factorial of %d = %d\n", num, factorial(num));
    return 0;
}
```

9. Arrays in C

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

Ans:
ARRAY IN C:
An **array** in C is a collection of elements of the same data type stored in contiguous memory locations. You access each element through its index using square bracket notation. Indexing starts at 0

One-Dimensional (1D) Arrays
Definition & Usage
A 1D array is essentially a linear sequence of elements—like a list or vector.
Declaration syntax:

```
data_type array_name[size];
```

Multi-Dimensional Arrays (e.g., 2D & 3D)
What Are They?
Multi-dimensional arrays are arrays of arrays—useful for structured data like matrices or higher-dimension datasets.
Two-Dimensional (2D) Arrays
Declaration:

```
data_type name[rows][columns];
```

 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

SINGAL AND MULTI DIMENSIONAL ARRAY:

```
#include<stdio.h>

int main() {
    //this is one dimensional array
    int arr[5];
    // Taking input from user
    printf("Enter 5 integers:\n");
    for (int i = 0; i < 5; i++) {
        scanf("%d", &arr[i]);
    }
    // Displaying the values
    printf("You entered:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d\n", arr[i]);
    }
    return 0;

    //this is multi dimensional array
    int arr2D[3][3];
    printf("Enter 9 integers for a 3x3 matrix:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            scanf("%d", &arr2D[i][j]);
        }
    }
    // Displaying the values
    printf("You entered:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr2D[i][j]);
        }
        printf("\n");

    }
    //sum of all values
    int sum = 0;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            sum += arr2D[i][j];
        }
    }
    printf("Sum of all values = %d\n", sum);
}
```

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?

Ans:

POINTER IN C:

A pointer in C is a variable that stores the memory address of another variable, rather than the variable's value. Dereferencing the pointer gives access to the value located at that address. Pointers enable indirect manipulation of data and form the backbone of low-level memory operations.

How to Declare and Initialize Pointers

Declaration

Use an asterisk * alongside the data type to declare a pointer:

```
int *ptr;
```

This means ptr can hold the address of an int variable.

Initialization

To initialize, use the address-of operator &:

```
int var = 10;
int *ptr = &var;
```

Now ptr stores the address of var.

 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
POINTER:

```
#include<stdio.h>

int main(){
   int num = 10;
   int *ptr = &num;  // Pointer variable
   printf("Value of num = %d\n", num);
   printf("Value at ptr = %d\n", *ptr);
   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.
Ans:
Overview of Common String Functions in C
All of these functions are part of the C standard library <string.h>, making string manipulation straightforward and efficient GeeksforGeeksdrbtaneja.com.
1. strlen()

- **Purpose**: Returns the length of a string, *excluding* the terminating null character ('\0').
- **Syntax**: size_t strlen(const char *str);
- **Example**:

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

int main() {
   char s[] = "Gfg";
   printf("%zu\n", strlen(s));  // Output: 3
   return 0;
}
```

- **Use-case**: Great for determining buffer sizes or iterating through strings when knowing their exact length is essential.

---

2. strcpy()

- **Purpose**: Copies the entire string (including the null character) from src to dest.
- **Syntax**: char *strcpy(char *dest, const char *src);
- **Example**:

```
char src[] = "Hello";
char dest[20];
strcpy(dest, src);
printf("%s\n", dest);  // Output: Hello
```

- **Caution**: Must ensure dest has enough space; otherwise, buffer overflows can occur Reddit.

---

3. strcat()

- **Purpose**: Appends (src) to the end of (dest), overwriting the null terminator.

- **Syntax**: char *strcat(char *dest, const char *src);
- **Example**:
  char s1[30] = "Hello, ";
  char s2[] = "Geeks!";
  strcat(s1, s2);
  printf("%s\n", s1); // Output: Hello, Geeks!

- **Use-case**: Useful when building longer strings from known parts—but again, watch buffer sizes!

---

4. strcmp()

- **Purpose**: Lexicographically compares two strings.
- **Syntax**: int strcmp(const char *s1, const char *s2);
- **Return Values**:
  - 0 → equal
  - < 0 → s1 is lexically smaller
  - > 0 → s1 is lexically greater
- **Example**:
  char s1[] = "Apple";
  char s2[] = "Applet";
  int res = strcmp(s1, s2);
  if (res == 0) printf("Same\n");
  else if (res < 0) printf("s1 is smaller\n");
  else printf("s1 is greater\n");

- **Use-case**: Ideal for sorting, comparisons, and validating string equality.

---

5. strchr()

- **Purpose**: Finds the *first* occurrence of a character in a string.
- **Syntax**: char *strchr(const char *str, int c);
- **Returns**: Pointer to the found character, or NULL if not found.
- **Example**:
  char s[] = "Hello, World!";
  char *res = strchr(s, 'o');
  if (res != NULL) printf("Found at index: %ld\n", res - s);
  else printf("Not found\n");

- **Use-case**: Helps locate characters for parsing, tokenizing, or conditional logic.

 LAB EXERCISE:
o 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>

int main(){
   // use of strcat()
   char str1[20] = "Hello ";
   char str2[20] = "World!";
   strcat(str1, str2);
   printf("%s\n", str1);
   return 0;

   // use of strlen()
```

```
    char str3[20] = "Hello World!";
    int len = strlen(str3);
    printf("Length of '%s' is %d\n", str3, len);
}
```

12. Structures in C

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

Ans:

STRUTURE IN C:

A **structure** (struct) in C is a user-defined, composite data type that allows you to group together variables of different types under one name—ideal for modeling real-world entities. For instance, a Student struct could include int, char[], and float members.

Declaring a Structure

General syntax:

```
struct TagName {
    data_type member1;
    data_type member2;
    // ...
};
```

Example:

```
struct Student {
    int roll;
    char name[50];
    float marks;
};
```

Now you can create variables of this type:

```
struct Student s1, s2;
```

You can also define and declare in one go:

```
struct Student {
    int roll;
    char name[50];
    float marks;
} s1, s2;
```

Initializing Structures

1. Using an initializer list (C89 style)

```
struct Student s1 = {101, "Aniket", 88.5};
```

Members are initialized in the order of declaration.

2. Using designated initializers (C99+)

```
struct Student s2 = { .marks = 92.0, .name = "Rajat", .roll = 102 };
```

This lets you initialize members out of order or only some of them. Uninitialized members default to zero.

3. Memberwise assignment after declaration

```
struct Student s3;
s3.roll = 103;
strcpy(s3.name, "Rahul");
s3.marks = 75.0;
```

Note: You cannot initialize members directly inside the struct definition—only when declaring a variable.

Accessing Structure Members

- **Dot operator (.):** Access via variable
  s1.roll = 105;
  printf("%s", s1.name);

- **Arrow operator (->):** Access via pointer
  struct Student *p = &s1;
  p->marks = 90.0;

Unstop

Copying Structures

You can copy one structure variable to another with simple assignment:
struct Student s2 = s1;
This performs a **shallow copy**, copying all member values.

 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>

int main(){
   struct Student {
      char name[50];
      int age;
      float marks;
   };

   struct Student s1;

   // Input student details
   printf("Enter name: ");
   fgets(s1.name, sizeof(s1.name), stdin);
   printf("Enter age: ");
   scanf("%d", &s1.age);
   printf("Enter marks: ");
   scanf("%f", &s1.marks);

   // Display student details
   printf("\nStudent Details:\n");
   printf("Name: %s", s1.name);
   printf("Age: %d\n", s1.age);
   printf("Marks: %.2f\n", s1.marks);

   return 0;

   //print using array
   struct Student students[100];
   int n;
```

```
    printf("Enter number of students: ");
    scanf("%d", &n);

    for(int i = 0; i < n; i++) {
        printf("Enter details for student %d:\n", i + 1);
        printf("Name: ");
        getchar();  // consume newline
        fgets(students[i].name, sizeof(students[i].name), stdin);
        printf("Age: ");
        scanf("%d", &students[i].age);
        printf("Marks: ");
        scanf("%f", &students[i].marks);
    }

    printf("\nStudent Details:\n");
    for(int i = 0; i < n; i++) {
        printf("Name: %s", students[i].name);
        printf("Age: %d\n", students[i].age);
        printf("Marks: %.2f\n", students[i].marks);
    }

    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.

Ans:

File Handling Matters in C

- **Persistent Storage**: Unlike variables stored in RAM that vanish when a program exits, file handling enables you to **store data permanently** on disk—whether it's user inputs, logs, or program state.
- **Large Data Management**: When dealing with massive datasets, loading everything into memory isn't practical. Files let you process data efficiently in chunks.
- **Automation & Logging**: Generate logs, reports, and maintainable output over time, especially useful in long-running tasks or debugging.
- **Data Interchange & Backup**: Files facilitate data sharing across programs and platforms, and support data recovery.

How to Handle Files in C: Step by Step

1. Opening a File

Use fopen() from <stdio.h> to open or create files, returning a FILE * pointer:

```
FILE *fptr = fopen("example.txt", "r");
```

- If opening fails, fopen() returns NULL. Always check this to prevent errors.
- **Modes** include:
    - "r" – read (file must exist)
    - "w" – write (creates/truncates file)
    - "a" – append
    - "r+", "w+", "a+" – combos for both reading and writing
    - Add "b" for binary mode (e.g., "rb", "wb+")

2. Writing to a File

You have several methods to write data:

- **Text output**:
  - fprintf(fptr, format, ...) – like printf() but to a file.
  - fputs(string, fptr) or fputc(char, fptr) – write strings or single characters.
- **Binary writing**:
  - fwrite(ptr, size, count, fptr) – writes raw bytes, useful for writing structs or arrays.

3. Reading from a File

Similarly, you can read in different ways:

- **Text input**:
  - fscanf(fptr, format, ...) – formatted input.
  - fgets(buffer, size, fptr) – reads a line into a buffer.
- **Binary reading**:
  - fread(ptr, size, count, fptr) – reads raw bytes into memory.

4. Closing a File

Always close files with fclose(fptr) when done:

- It **flushes any buffered data** and **releases system resources**.
- Returns 0 on success, or EOF on failure.

Putting It All Together: Example Code

Example 1: Simple Text File Write and Read

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

int main() {
    FILE *fptr = fopen("data.txt", "w");
    if (!fptr) {
        perror("Cannot open file");
        return 1;
    }
    fprintf(fptr, "Hello, C file handling!\n");
    fclose(fptr);  // Always close file

    fptr = fopen("data.txt", "r");
    if (!fptr) {
        perror("Cannot open file");
        return 1;
    }
    char buf[100];
    while (fgets(buf, sizeof(buf), fptr)) {
        printf("Read: %s", buf);
    }
    fclose(fptr);
    return 0;
}
```

Example 2: Writing and Reading a Struct as Binary

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

struct Pair { int x, y; };

int main() {
    struct Pair p = {1, 2};
    FILE *fptr = fopen("pair.bin", "wb");
    if (!fptr) { perror("Open failed"); return 1; }
    fwrite(&p, sizeof(p), 1, fptr);
    fclose(fptr);
```

```
    struct Pair q;
    fptr = fopen("pair.bin", "rb");
    if (!fptr) { perror("Open failed"); return 1; }
    fread(&q, sizeof(q), 1, fptr);
    fclose(fptr);
    printf("Read pair: %d, %d\n", q.x, q.y);

    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.

Ans

FILE HANDILING:

```
#include<stdio.h>

int main(){
    //open a file
    FILE *file = fopen("data.txt", "r");
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    //read from the file
    char buffer[100];
    while (fgets(buffer, sizeof(buffer), file)) {
        printf("%s", buffer);
    }

    //close the file
    fclose(file);

    //reopen and display
    file = fopen("data.txt", "r");
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    //read from the file
    while (fgets(buffer, sizeof(buffer), file)) {
        printf("%s", buffer);
    }

    //close the file
    fclose(file);
    return 0;
}
```

EXTRA LAB EXERCISES FOR IMPROVING PROGRAMMING LOGIC

1. Operators

LAB EXERCISE 1: Simple Calculator

 Write a C program that acts as a simple calculator. The program should take two numbers and an operator as input from the user and perform the respective operation (addition, subtraction, multiplication, division, or modulus) using operators.

 Challenge: Extend the program to handle invalid operator inputs.

Ans

```
#include <stdio.h>

int main() {
   float num1, num2;
   char op;
   printf("Enter two numbers: ");
   scanf("%f %f", &num1, &num2);
   printf("Enter operator (+, -, *, /, %%): ");
   scanf(" %c", &op);

   switch(op) {
      case '+': printf("Result: %.2f\n", num1 + num2); break;
      case '-': printf("Result: %.2f\n", num1 - num2); break;
      case '*': printf("Result: %.2f\n", num1 * num2); break;
      case '/':
         if(num2 != 0)
            printf("Result: %.2f\n", num1 / num2);
         else
            printf("Error: Division by zero!\n");
         break;
      case '%':
         if((int)num2 != 0)
            printf("Result: %d\n", (int)num1 % (int)num2);
         else
            printf("Error: Modulo by zero!\n");
         break;
      default: printf("Invalid operator!\n");
   }
   return 0;
}
```

LAB EXERCISE 2: Check Number Properties

 Write a C program that takes an integer from the user and checks the following using different operators:

- Whether the number is even or odd.
- Whether the number is positive, negative, or zero.
- Whether the number is a multiple of both 3 and 5.

Ans

```
#include <stdio.h>

int main() {
   int num;
   printf("Enter an integer: ");
   scanf("%d", &num);

   // Even or Odd
   if (num % 2 == 0) printf("Even\n");
   else printf("Odd\n");

   // Positive, Negative, or Zero
   if (num > 0) printf("Positive\n");
   else if (num < 0) printf("Negative\n");
   else printf("Zero\n");

   // Multiple of 3 and 5
   if (num % 3 == 0 && num % 5 == 0) printf("Multiple of 3 and 5\n");
   else printf("Not a multiple of both 3 and 5\n");

   return 0;
}
```

2. Control Statements

LAB EXERCISE 1: Grade Calculator

 Write a C program that takes the marks of a student as input and displays the corresponding grade based on the following conditions:

- Marks > 90: Grade A
- Marks > 75 and <= 90: Grade B
- Marks > 50 and <= 75: Grade C
- Marks <= 50: Grade D

 Use if-else or switch statements for the decision-making process.

Ans

```
#include <stdio.h>

int main() {
   int marks;
   printf("Enter marks: ");
   scanf("%d", &marks);

   if (marks > 90)
      printf("Grade A\n");
   else if (marks > 75)
      printf("Grade B\n");
   else if (marks > 50)
```

```
    printf("Grade C\n");
  else
    printf("Grade D\n");

  return 0;
}
```

LAB EXERCISE 2: Number Comparison

- Write a C program that takes three numbers from the user and determines:
  - The largest number.
  - The smallest number.
- **Challenge**: Solve the problem using both if-else and switch-case statements.

Ans

```
#include <stdio.h>

int main() {
  int a, b, c;
  printf("Enter 3 numbers: ");
  scanf("%d %d %d", &a, &b, &c);

  int largest = a, smallest = a;

  if (b > largest) largest = b;
  if (c > largest) largest = c;

  if (b < smallest) smallest = b;
  if (c < smallest) smallest = c;

  printf("Largest: %d\n", largest);
  printf("Smallest: %d\n", smallest);

  return 0;
}
```

3. Loops

LAB EXERCISE 1: Prime Number Check

- Write a C program that checks whether a given number is a prime number or not using a for loop.
- **Challenge**: Modify the program to print all prime numbers between 1 and a given number.

Ans

```
#include <stdio.h>

int main() {
  int n, i, flag = 1;
  printf("Enter a number: ");
  scanf("%d", &n);

  if (n <= 1) flag = 0;

  for (i = 2; i <= n/2; ++i) {
    if (n % i == 0) {
      flag = 0;
      break;
```

```
        }
    }

    if (flag)
        printf("%d is a prime number.\n", n);
    else
        printf("%d is not a prime number.\n", n);

    return 0;
}
```

LAB EXERCISE 2: Multiplication Table

 Write a C program that takes an integer input from the user and prints its multiplication table using a for loop.

 Challenge: Allow the user to input the range of the multiplication table (e.g., from 1 to N).

Ans

```
#include <stdio.h>

int main() {
    int n, i, range;
    printf("Enter a number: ");
    scanf("%d", &n);
    printf("Enter range: ");
    scanf("%d", &range);

    for (i = 1; i <= range; i++) {
        printf("%d x %d = %d\n", n, i, n * i);
    }

    return 0;
}
```

LAB EXERCISE 3: Sum of Digits

 Write a C program that takes an integer from the user and calculates the sum of its digits using a while loop.

 Challenge: Extend the program to reverse the digits of the number.

Ans

```
#include <stdio.h>

int main() {
   int num, sum = 0, rev = 0, digit;
   printf("Enter a number: ");
   scanf("%d", &num);
   int temp = num;

   while (num > 0) {
      digit = num % 10;
      sum += digit;
      rev = rev * 10 + digit;
      num /= 10;
   }

   printf("Sum of digits: %d\n", sum);
   printf("Reversed number: %d\n", rev);

   return 0;
}
```

4. Arrays

LAB EXERCISE 1: Maximum and Minimum in Array

 Write a C program that accepts 10 integers from the user and stores them in an array. The program should then find and print the maximum and minimum values in the array.

 Challenge: Extend the program to sort the array in ascending order.

Ans

```
#include <stdio.h>

int main() {
   int arr[10], i, max, min;

   printf("Enter 10 numbers:\n");
   for (i = 0; i < 10; i++) scanf("%d", &arr[i]);

   max = min = arr[0];
   for (i = 1; i < 10; i++) {
      if (arr[i] > max) max = arr[i];
      if (arr[i] < min) min = arr[i];
   }

   printf("Max: %d, Min: %d\n", max, min);

   // Sorting
   for (i = 0; i < 9; i++) {
      for (int j = i+1; j < 10; j++) {
         if (arr[i] > arr[j]) {
```

```
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
  }

  printf("Sorted Array: ");
  for (i = 0; i < 10; i++) printf("%d ", arr[i]);
  return 0;
}
```

LAB EXERCISE 2: Matrix Addition

 Write a C program that accepts two 2x2 matrices from the user and adds them. Display the resultant matrix.

 Challenge: Extend the program to work with 3x3 matrices and matrix multiplication.

Ans

```
#define SIZE 2
#include <stdio.h>

int main() {
  int a[SIZE][SIZE], b[SIZE][SIZE], sum[SIZE][SIZE];

  printf("Enter elements of matrix A:\n");
  for(int i = 0; i < SIZE; i++)
    for(int j = 0; j < SIZE; j++)
      scanf("%d", &a[i][j]);

  printf("Enter elements of matrix B:\n");
  for(int i = 0; i < SIZE; i++)
    for(int j = 0; j < SIZE; j++)
      scanf("%d", &b[i][j]);

  printf("Sum matrix:\n");
  for(int i = 0; i < SIZE; i++) {
    for(int j = 0; j < SIZE; j++) {
      sum[i][j] = a[i][j] + b[i][j];
      printf("%d ", sum[i][j]);
    }
    printf("\n");
  }

  return 0;
}
```

LAB EXERCISE 3: Sum of Array Elements

 Write a C program that takes N numbers from the user and stores them in an array. The program should then calculate and display the sum of all array elements.

 Challenge: Modify the program to also find the average of the numbers.

Ans

```
#define SIZE 2
#include <stdio.h>
```

```
int main() {
   int a[SIZE][SIZE], b[SIZE][SIZE], sum[SIZE][SIZE];

   printf("Enter elements of matrix A:\n");
   for(int i = 0; i < SIZE; i++)
      for(int j = 0; j < SIZE; j++)
         scanf("%d", &a[i][j]);

   printf("Enter elements of matrix B:\n");
   for(int i = 0; i < SIZE; i++)
      for(int j = 0; j < SIZE; j++)
         scanf("%d", &b[i][j]);

   printf("Sum matrix:\n");
   for(int i = 0; i < SIZE; i++) {
      for(int j = 0; j < SIZE; j++) {
         sum[i][j] = a[i][j] + b[i][j];
         printf("%d ", sum[i][j]);
      }
      printf("\n");
   }

   return 0;
}
```

5. Functions

LAB EXERCISE 1: Fibonacci Sequence

 Write a C program that generates the Fibonacci sequence up to N terms using a recursive function.

 Challenge: Modify the program to calculate the Nth Fibonacci number using both iterative and recursive methods. Compare their efficiency.

Ans

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n - 1) + fib(n - 2);
}
```

LAB EXERCISE 2: Factorial Calculation

 Write a C program that calculates the factorial of a given number using a function.

 Challenge: Implement both an iterative and a recursive version of the factorial function and compare their performance for large numbers.

Ans

```
long factorial(int n) {
   if (n <= 1) return 1;
   return n * factorial(n - 1);
}
```

LAB EXERCISE 3: Palindrome Check

 Write a C program that takes a number as input and checks whether it is a palindrome using a function.

 Challenge: Modify the program to check if a given string is a palindrome.

Ans

```
int is_palindrome(int n) {
   int rev = 0, temp = n;
   while (n != 0) {
      rev = rev * 10 + n % 10;
      n /= 10;
   }
   return rev == temp;
}
```

6. Strings

LAB EXERCISE 1: String Reversal

 Write a C program that takes a string as input and reverses it using a function.

 Challenge: Write the program without using built-in string handling functions.

Ans

```
void reverse(char str[]) {
   int len = 0;
   while (str[len] != '\0') len++;

   for (int i = len - 1; i >= 0; i--)
      printf("%c", str[i]);
}
```

LAB EXERCISE 2: Count Vowels and Consonants

 Write a C program that takes a string from the user and counts the number of vowels and consonants in the string.

 Challenge: Extend the program to also count digits and special characters.

Ans

```
int vowels = 0, consonants = 0, digits = 0, special = 0;
```

LAB EXERCISE 3: Word Count

 Write a C program that counts the number of words in a sentence entered by the user.

 Challenge: Modify the program to find the longest word in the sentence.

Ans

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

int main() {
   char str[1000];
   int i, word_count = 0, in_word = 0;

   printf("Enter a sentence: ");
   fgets(str, sizeof(str), stdin);  // safer than gets()

   for (i = 0; str[i] != '\0'; i++) {
```

```
        if (isspace(str[i])) {
            in_word = 0;
        } else if (in_word == 0) {
            in_word = 1;
            word_count++;
        }
    }

    printf("Total number of words: %d\n", word_count);
    return 0;
}
```

Extra Logic Building Challenges

Lab Challenge 1: Armstrong Number

 Write a C program that checks whether a given number is an Armstrong number or not (e.g., 153 = 1^3 + 5^3 + 3^3).

 Challenge: Write a program to find all Armstrong numbers between 1 and 1000.

Ans

```
int is_armstrong(int n) {
    int sum = 0, temp = n;
    while (n != 0) {
        int d = n % 10;
        sum += d * d * d;
        n /= 10;
    }
    return sum == temp;
}
```

Lab Challenge 2: Pascal's Triangle

 Write a C program that generates Pascal's Triangle up to N rows using loops.

 Challenge: Implement the same program using a recursive function.

Ans

```
#include <stdio.h>

// Function to calculate factorial
int factorial(int n) {
    int f = 1;
    for (int i = 1; i <= n; i++)
        f *= i;
    return f;
}

// Function to calculate nCr (combination)
int combination(int n, int r) {
    return factorial(n) / (factorial(r) * factorial(n - r));
}

int main() {
    int rows;
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
```

```
   for (int i = 0; i < rows; i++) {
      // Print spaces for formatting
      for (int space = 0;
```

Lab Challenge 3: Number Guessing Game

 Write a C program that implements a simple number guessing game. The program should generate a random number between 1 and 100, and the user should guess the number within a limited number of attempts.

 Challenge: Provide hints to the user if the guessed number is too high or too low.

Ans

```
#include <stdlib.h>
#include <time.h>
int main() {
   srand(time(0));
   int num = rand() % 100 + 1, guess, attempts = 0;

   while (attempts < 7) {
      printf("Guess the number (1-100): ");
      scanf("%d", &guess);
      attempts++;

      if (guess == num) {
         printf("Correct! You won in %d attempts.\n", attempts);
         break;
      } else if (guess > num) {
         printf("Too high!\n");
      } else {
         printf("Too low!\n");
      }
   }

   if (guess != num)
      printf("Out of attempts! Number was %d\n", num);
}
```