Module 2 – Introduction to Programming

Name : Chandvaniya Abhay

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.

- The history and evolution of C programming:

➢ The story of the C programming language begins in the early 1970s, when computers were large, complex machines, and programming was very different than today. **Dennis Ritchie**, working at **Bell Labs**, decided to create a language that was efficient, flexible, and powerful enough to help build the **Unix operating system**.

➢ Before C, there was a programming language called **B**, which itself was derived from an earlier language called **BCPL**. These languages helped shape the ideas behind C. But B lacked features and efficiency needed to fully harness the capabilities of the then new PDP-11 computer.

➢ Dennis Ritchie developed C in **1972** as a system programming language capable of replacing assembly language, giving programmers more control over hardware while still being easier to write and understand.

Year	Version	Notable Events
1972	First release	Dennis Ritchie creates C for Unix development.
1978	K&R C	"The C Programming Language" by Kernighan & Ritchie sets a standard.
1989	ANSI C/C89	First official ANSI standardization of the language.
1990	ISO C/C90	Minor updates, international standard recognized.
1999	C99	Adds features like inline functions & new data types.
2011	C11	Multi-threading, safer functions, Unicode support.
2018	C18	Minor updates and corrections.
2024	C23	Latest standard

- Over time, C was standardized by **ANSI** and **ISO** committees, evolving with new features over the decades but never losing

the simplicity and speed that made it so popular in the first place.

- The Importance of C Programming**:**

➢ C is often referred to as the "mother of all languages" because it influenced the creation of many prominent languages, including C++, Java, and Python.

➢ C gives programmers close control over how a computer's memory is managed and how its hardware is used, so software can be very precisely tuned for performance. Higher-level languages usually add extra layers for convenience, but C lets developers shape programs exactly as needed, without much overhead.

➢ **Portability**: Unlike assembly language, which is tied to a specific processor, C code can be compiled and run almost anywhere with minimal changes. This ability to "**write once, run anywhere**" was revolutionary in an era of diverse hardware.

➢ Learning C provides a deep, clear understanding of what happens under the hood of more modern, high-level languages. For example, if you know how C uses pointers and manages memory, concepts in languages like Python, Java, or C++ often make much more sense.

➢ **Vast Ecosystem and Legacy Code:** Since C has been around for so long, a huge amount of software has been written in it. Critical parts of operating systems like Windows, Linux, and macOS, as well as database systems, network drivers, and many embedded systems, are written in C.

- why it is still used today:

- In today's programming world filled with fancy high-level languages and frameworks you might wonder if learning C is still worthwhile. The answer is absolutely yes. Whether it's for understanding how software really works, developing software that needs to run close to the hardware, or maintaining and improving the massive legacy code bases still in use, C remains an essential skill for programmers.

- Crucial for real-time, embedded, and high-performance domains like databases, graphics engines, and AI frameworks. Often the core processing engines of modern machine learning frameworks are written in C for speed.

- Its combination of speed, control, simplicity, and portability helped revolutionize computing and created the foundation for countless technologies we rely on every day. That's why, even after more than fifty years, C programming continues to matter.

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.

- Operating Systems:

➢ Operating systems like Windows, Linux, and macOS are mostly built with C. C lets developers work close to the hardware, controlling memory and system resources efficiently. This means the OS can run fast and be stable. For example, the Linux kernel, which powers everything from servers to Android phones, is almost entirely written in C. This helps the OS be portable and work on different types of machines.

➢ C language's strengths are efficient memory management, direct hardware access via pointers, and deterministic execution make it ideal for low-level system components that require tight control and high performance. As a result, much of the development of file systems, device drivers, networking stacks, and process schedulers in these platforms relies on C.

- Embedded Systems:

➢We use embedded systems daily without noticing your microwave, car braking systems, or fitness trackers. These tiny computers inside devices need to be super efficient since they often have limited memory and processing power. C is perfect for this because it allows programmers to write lean code that talks directly to hardware parts, like sensors or motors.

➢ Since C is fast and doesn't require much memory, it's perfect for these resource-limited devices. C programming is often running behind the scenes. Embedded systems are small computers built into larger machines to control their functions.

➢For ex- automotive systems that manage tasks like anti-lock braking or airbag deployment rely on code written in C because it needs to be ultra-reliable, quick, and capable of interacting closely with the vehicle's hardware

- <u>Game Development:</u>

- In the early days of video games, memory and processing power were extremely limited, so developers turned to C lang to squeeze every bit of performance possible out of consoles and PCs.

- This gave developers precise control over how graphics, sound, and gameplay were managed. Even in modern times, many engines have their core, the parts that do the real heavy lifting for things like physics and graphics rendering implemented in C (or its close cousin, C++) for efficiency. C's direct hardware control and speed make it possible.

2. 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.

- Step to install GCC Compiler:-

Step 1: Download MinGW-w64

Go to the official MinGW-w64 releases or trusted source for MinGW-w64.

Download the appropriate installer

Step 2: Extract and Set Up

Extract the archive to a simple directory, e.g., C:\mingw64.

Step 3: Add MinGW-w64 to PATH

Open "Environment Variables" (under System Properties).

Edit the Path variable and add the path to the bin folder, e.g., C:\mingw64\bin.

Step 4: Verify Installation

Open Command Prompt and run: gcc --version

If installed correctly, it will display the GCC version 1.

- Steps for installing and Setup DevC++

Step 1: Go to the official DevC++ website

Step 2: Download the latest executable installer.

Step 3: Run the installer.

Step 4: Follow the installation wizard, agreeing to license and selecting default components.

Step 5: Specify the install path and let it finish.

Step 6: Launch DevC++ and, if necessary, configure the compiler path. DevC++ often comes bundled with MinGW for basic C/C++ usage.

- Steps for installing and Set Up Visual Studio Code (VS Code)

Step 1: Visit the official VS Code website and download the Windows installer.

Step 2: Run the installer, accept terms, select install location, and check optional tasks (like "Add to PATH", context menu integration)

Step 3: Finish installation and launch VS Code.

Step 4: Install the "C/C++" extension from Microsoft (available from the Extensions panel in VS Code).

Step 5: Ensure GCC/MinGW or any C compiler is installed (see above).

Step 6: Configure tasks and debugging:

Create a tasks.json to allow code compilation within VS Code.

Set up launch.json for debugging if needed.

- Steps for installing and Set Up CodeBlocks IDE

Step 1: Visit the official CodeBlocks downloads page.

Step 2: Choose the *"codeblocks-XX.Xmingw-setup.exe"*
(includes both CodeBlocks IDE and MinGW compiler).

Step 3: Download and run the installer.

Step 4: Follow the on-screen instructions, accepting license and leaving defaults.

Step 5: Once installed, open CodeBlocks.

Step 6: The IDE should detect the bundled compiler. If not, go to Settings > Compiler, select the correct MinGW path, and save.

LAB EXERCISE:

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

➢ For the C compiler I installed MinGW, which provides the GCC (GNU Compiler Collection) for Windows.

➢ program to print "Hello, World!"

```
#include <stdio.h>

int main()
 {
   printf("\nHello World");
   return 0;
 }
```

3. 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.

- Headers:

➢ Headers are files included at the beginning of a C program to provide declarations for functions and macros.

#include <stdio.h>

'#'- pre processor

'.h'- header file
'Stdio'-library header file

- Main Function:

➢ Starts program execution

```
int main()  / void main() //depends on library
 {
   // program code
   return 0;
```

}

- ## Comments:

➢ To improve code readability and are ignored by the compiler. They come in two forms:

// This is a single-line comment

/* This is a multi-line comment */

- ## Data Types:

➢ Define variable types. C has several built-in data types,

➔ int (integer): whole numbers (e.g., 1, 2, 3)
➔ float (floating point):decimal numbers (e.g., 3.14, -0.5)
➔ double (double-precision floating point)
➔ char (character):single characters (e.g., 'a', 'B')

Ex-

```
#include<stdio.h>

int main()

{
```

```
int age = 25;

float height = 6.2;

double balance = 12345.67;

char initial = 'H';


printf(" \nMy age is : %d ", age);

printf(" \nMy height is : %f" , height);

printf(" \nMy balance is : %lf ", balance);

printf(" \nMy initial is : %c", initial);


return 0;

}
```

- Variables:-

➢ Variables are used to store data. You must declare the variable type before using it.

Ex-

```
int a;          // Declaration
a = 10;         // Initialization
float b = 3.14;   // Declaration + Initialization
```

➢ Here's an example program that demonstrates these concepts-

```c
#include <stdio.h>       // Header file for input/output

/* This program demonstrates basic structure of C */

int main()      // Main function
{
  // Variable declarations and initialization
   int age = 20;
   float percentage = 87.5;
   char grade = 'A';

   // Output values
   printf("\nAge :  %d", age);
   printf("\nPercentage :  %f", percentage);
   printf("\nGrade :  %c", grade);

   return 0;
}
```

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.

```c
#include <stdio.h>


float PI=3.14159;


int main()
{
                    // Declare an integer variable
   int age = 22;


                     // Declare a character variable
   char initial = 'A';


   // Declare a float variable and use the PI constant
   float circleRadius = 2.2;
   float circleArea = PI * circleRadius * circleRadius;


   // Print all values to the console
   printf("\nAge : %d", age);          // Display integer value
```

```
    printf("\nInitial : %c", initial);      // Display character value
    printf("Circle radius  :  %f\n", circleRadius); // Display float value with 2 decimal places
    printf("Circle area : %f\n", circleArea);     // Display calculated float value

    // Printing the constant
    printf("Value of PI  :  %f\n", PI);

    // Return success
    return 0;
}
```

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.

- Arithmetic Operators:-

➢ Used to perform mathematical operations.

➢ Operators: + (Addition),
- (Subtraction),
* (Multiplication),
/ (Division),
% (Modulus).

Example: a + b, a % b.

➢ Purpose: Calculate values like sums, differences, products, quotients, and remainders.

- Relational Operators:-

➢ Used to compare two values or expressions.
➢ Operators: == (Equal to),
!= (Not equal to),
> (Greater than),
< (Less than),
>= (Greater than or equal to),
<= (Less than or equal to).

Example: a > b

➢ Purpose: Returns either true (1) or false (0) for use in conditions.

- Logical Operators:-

➢ Used to combine or invert conditions.
➢ Operators:

&& (Logical AND): True if both operands are true.

|| (Logical OR): True if at least one operand is true.

! (Logical NOT): True if the operand is false.

- Assignment Operators:

- Used to assign values to variables.
- Operators: = (Simple assignment), +=, -=, *=, /=, %= (Compound assignments).

  Example: a = 5, a += 3 (equivalent to a = a + 3)

- Purpose: Store the result of an expression in a variable.

• Increment/Decrement Operators:-

- Used to increase or decrease a variable's value by one.
- Operators: ++ (Increment),

  -- (Decrement),

  ++a, a++(used as prefix or postfix)
- Purpose: Commonly used in loops and counting

• Bitwise Operators:-

- Work at the binary level on integer data.
- Operators:

  & (AND), | (OR), ^ (XOR), ~ (NOT)

  << (Left shift), >> (Right shift)

  Example: a & b, a << 2

- Purpose: Manipulate data at bit level, useful in system programming.

• Conditional (Ternary) Operator:-

- Short form for simple if-else statements.

- Operator:

  ?: (ternary operator)

  Syntax: condition ? value_if_true : value_if_false

  Example: max = (a > b) ? a : b;

- Purpose: Conditionally assign a value or execute code in a concise way.

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.

```c
#include <stdio.h>


int main()
{
   int a, b;



   printf("Enter first integer: ");
   scanf("%d", &a);

   printf("Enter second integer: ");
   scanf("%d", &b);



   printf("\nArithmetic Operations : ");
   printf("\n%d  +  %d = %d", a, b, a + b);
   printf("\n%d  -  %d = %d", a, b, a - b);
   printf("\n%d  *  %d = %d", a, b, a * b);
```

```
    printf("\n%d  /  %d = %d", a, b, a / b);
    printf("\n%d  %%  %d = %d", a, b, a % b);




    printf("\nRelational Operations : ");
    printf("\n%d == %d  :  %s", a, b, (a == b) ? "True" : "False");
    printf("\n%d != %d : %s", a, b, (a != b) ? "True" : "False");
    printf("\n%d > %d  : %s", a, b, (a > b) ? "True" : "False");
    printf("\n%d < %d  : %s", a, b, (a < b) ? "True" : "False");
    printf("\n%d >= %d : %s", a, b, (a >= b) ? "True" : "False");
    printf("\n%d <= %d : %s", a, b, (a <= b) ? "True" : "False");



    printf("\nLogical Operations (considering non-zero as true and zero as false) :
");
    printf("\n(%d && %d)  :  %s", a, b, (a && b) ? "True" : "False");
    printf("\n(%d || %d)  :  %s", a, b, (a || b) ? "True" : "False");
    printf("\n!%d : %s", a, (!a) ? "True" : "False");
    printf("\n!%d : %s", b, (!b) ? "True" : "False");

    return 0;
}
```

5. Control Flow Statements in C

THEORY EXERCISE:

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

➢ Decision-making statements in C allow you to execute different parts of code based on certain conditions. The primary decision-making statements in C are: if, else, nested if-else, and switch.

- if Statement:

➢ The if statement evaluates a condition. If it is true, the code block inside the if is executed.

Ex-

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

int number = 10;
if (number > 0)
 {
   printf("\nNumber is positive");
}
```

```
 getch();
}
```

- if-else Statement:

➢ Adds an alternative: if the condition is false, the else block is executed.

Ex-

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

   int marks = 35;

   if(marks>=35)
   {
      printf("Pass");
   }
   else
   {
      printf("Fail");
   }
    getch();
}
```

- Nested if-else Statement:-

➢ You can place an if or if-else inside another if or else block to test multiple conditions.

Ex-

```
#include<stdio.h>
#include<conio.h>
void main()
{
int number = 0;

if (number > 0)
{
   printf("Positive number\n");
}

else if (number < 0)
{
   printf("Negative number\n");
}

else
{
   printf("Number is zero\n");
}

getch();
}
```

- switch Statement:

➢ The switch statement is used to select one of many code blocks to execute based on the value of a variable.

Ex-

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int num;

   printf("Enter Your Choice: ");
   scanf("%d",&num);//1


   switch(num)
   {
     case 1:printf("\n Your Selected Language is English");
     break;

     case 2:printf("\n Your Selected Language is Hindi");
     break;

     case 3:printf("\n Your Selected Language is Gujarati");
     break;

     default:printf("\n Your Number is not Valid");
     break;
```

```
  }


   getch();
}
```

Statement	Use
if	Executes block if condition is true
if-else	Executes block based on true/false condition
nested if-else	Multiple conditions checked in order
switch	Chooses block to execute based on variable's value

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.).

- display the month name based on the user's input-

```c
#include <stdio.h>

int main()
{
   int number, month;
```

```
   // Check if a number is even or odd
   printf("Enter an integer number: ");
   scanf("%d", &number);

   if (number % 2 == 0)
      {
        printf("%d is Even.\n", number);
      }
    else
         {
           printf("%d is Odd.\n", number);
         }

//-----------------------------------------------------
   // Display month name based on user input

   printf("Enter month number (1 to 12) : ");
   scanf("%d", &month);

   switch (month)

    {
case 1:
         printf("Month is January\n");
         break;
      case 2:
         printf("Month is February\n");
         break;
      case 3:
         printf("Month is March\n");
```

```
        break;
    case 4:
        printf("Month is April\n");
        break;
    case 5:
        printf("Month is May\n");
        break;
    case 6:
        printf("Month is June\n");
        break;
    case 7:
        printf("Month is July\n");
        break;
    case 8:
        printf("Month is August\n");
        break;
    case 9:
        printf("Month is September\n");
        break;
    case 10:
        printf("Month is October\n");
        break;
    case 11:
        printf("Month is November\n");
        break;
    case 12:
        printf("Month is December\n");
        break;
    default:
```

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

    return 0;
}
```

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.

- While Loops:

➢ Entry-controlled loop

➢ How it works:

-Tests the condition before executing the block.

-If the condition is false initially, the block may never

execute.

➢ Appropriate Scenarios:

-When the number of iterations is not predetermined.

-Useful for input-driven or sentinel-controlled loops, such as reading input until the end of file or until a special value is entered.

➢ Syntax:

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

- For Loop

➢ Entry-controlled loop.

➢ How it works:

-The most compact form for loops with a known or fixed number of iterations.

-Combines loop control (init, test, increment) all in the header.

➢ Appropriate Scenarios:

-When you know beforehand how many times to iterate.

Common for iterating simple counters, processing arrays, or running through indices.

➢ Syntax:

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

● Do-While Loop

➢ Exit-controlled loop.

➢ How it works:

-Executes the block **at least once** (since condition is checked after).

-The test occurs at the end of the loop.

➢ Appropriate Scenarios:

-When the loop must execute at least once regardless of the condition.

-Typical use-cases include menu-driven programs, retry-until-success patterns, and user confirmation dialogs.

➢ Syntax:

```
do
{
   // code block
```

```
}
 while (condition);
```

- Summary Table:

Loop Type	Condition Check	Minimum Iterations	Best For
while	before block	0	Unknown, input-driven loops, sentinel-controlled
for	before block	0	Known/fixed iterations, arrays, counting
do-while	after block	1	Must-run-once tasks, user/input validation, retries

- Choosing the Right Loop When deciding which loop to use, consider the following factors:

➢ Number of iterations: If you know the exact number, use a for loop. If it's unknown, use a while loop.

- ➢ Condition checking: If you want to check the condition before executing the loop body, use a while loop. If you want to execute the loop body at least once, use a do-while loop.

- ➢ Initialization: If you need to initialize variables before the loop, use a for loop or while loop.

LAB EXERCISE:

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

```c
#include <stdio.h>

int main()
{
   int i;

   // Using while loop
   printf("\nNumbers from 1 to 10 using while loop : ");
   i = 1;
   while (i <= 10)
  {
      printf("%d ", i);
      i++;
   }
   printf("\n");

   //-------------------------------------------------

   // Using for loop
   printf("\nNumbers from 1 to 10 using for loop : ");
   for (i = 1; i <= 10; i++)
   {
      printf("%d ", i);
   }
   printf("\n");
```

```c
    //-----------------------------------------------------

    // Using do-while loop
    printf("\nNumbers from 1 to 10 using do-while loop : ");
    i = 1;
    do
   {
        printf("%d ", i);
        i++;
    }
    while (i <= 10);
    printf("\n");

    return 0;
}
```

Output:-

Numbers from 1 to 10 using while loop : 1 2 3 4 5 6 7 8 9 10

Numbers from 1 to 10 using for loop : 1 2 3 4 5 6 7 8 9 10

Numbers from 1 to 10 using do-while loop : 1 2 3 4 5 6 7 8 9 10

7. Loop Control Statements

THEORY EXERCISE:

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

- break Statement:

➢ Use: The break statement is used to exit from a loop (for, while, do-while) or a switch statement before it would normally terminate.
➢ Typical Use Cases: When a condition is met and you want to stop execution of the loop or switch-case.

Example:

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

- continue Statement:

➢ Use: The continue statement skips the remaining code in the current loop iteration and moves to the next iteration.

➢ Typical Use Cases: When certain conditions are met and you want to skip to the next cycle of the loop without terminating the entire loop.

Example:

```c
#include <stdio.h>
int main()
{
   int i;
   for (i = 0; i < 5; i++)
{
      if (i == 2)
   {
         continue;
   }
      printf("%d ", i);
 }
     return 0;
}
```

- goto Statement:

➢ Use: The goto statement transfers control to the labeled statement in the same function.

➢ Typical Use Cases: Jumping out of deeply nested loops or error handling. Its use is discouraged as it can make code hard to follow and maintain.

Example:

```c
#include <stdio.h>
int main()
{
   int i = 0;
   start:
      printf("%d ", i);
      i++;
      if (i < 3)
         goto start;
   return 0;
}
```

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.

```c
// Exit the loop when i =5

#include <stdio.h>

int main()
{
   int i;
   for(i = 1; i <= 10; i++)
   {
      if(i == 5)
     {
         break;
      }
      printf("\n%d", i);
   }
   return 0;
}
```

Output-

1
2

3
4

```c
// Skip printing 3

#include <stdio.h>

int main()
{
   int i;
   for(i = 1; i <= 10; i++)
   {
      if(i == 5)
     {
         break;
      }

      if(i == 3)
      {
         continue;
      }
      printf("\n%d", i);
   }

   return 0;
}
```

Output-

1
2
4

8. Functions in C

THEORY EXERCISE:

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

➢ A function in C is a self-contained block of code that performs a specific task.

➢ Functions allow you to organize your code, promote reusability, and make large programs more manageable by dividing tasks into smaller sub-tasks.

➢ By using functions, programmers can reduce code repetition, improve readability, and make programs easier to maintain and debug.

- Types of functions:

1. Library (Predefined) Functions:

➢ These functions are already defined in the C standard library, so programmers do not have to write their implementation.
➢ These functions help perform common tasks like input/output operations, mathematical calculations, string manipulations, etc.
2. User-Defined Functions:

- ➢ These are functions that programmers define to perform specific tasks tailored to their needs.
- ➢ User-defined functions increase code reusability and modularity by encapsulating logic in one place.
- ➢ These must be declared, defined, and then called explicitly by the user in their program.

Type	Arguments	Return Value	Description
User-Defined Functions	Varies	Varies	Defined by the programmer
Function with no arguments, no return	No	No	Executes code without inputs or outputs
Function with no arguments, with return	No	Yes	Returns a value without inputs
Function with arguments, no return	Yes	No	Takes inputs but does not return a value
Function with arguments, with return	Yes	Yes	Takes inputs and returns a processed value

- ## Function Declaration:

➢ Also known as the function prototype, the declaration informs the compiler about the function's name, return type, and the types of its parameters.

➢ It acts as a forward declaration that allows the function to be called before it is actually defined in the code.

➢ The general syntax for a function declaration is:

```
return_type function_name(parameter_type1, parameter_type2, ...);
```

Ex-

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

- ## Function Definition:

➢ A function definition provides the actual body of the function, describing what the function does.

➢ The definition includes the function header (which repeats the return type, function name, and parameter list, including parameter names) and the function body enclosed within braces {}.

➢ A sample definition for the add function could be:

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

- Function Call:-

➢ The function call is you giving a command to use the function.

➢ Once called, the function runs its code, and if designed to, it sends back a result.

➢ Syntax:

function_name(argument1, argument2, ...);

➢ Example:

```
int result = add(8, 3);
// In this example, the add function is called with arguments 8 and 3, and the result is stored in the result variable.
```

- Complete Example:

```c
#include <stdio.h>

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

int main()
{
    int sum;
    sum = add(10, 20);   // Function call
    printf("Sum is %d\n", sum);
    return 0;
}

int add(int x, int y)      // Function definition

{
    return x + y;
}
```

LAB EXERCISE:

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

```c
#include <stdio.h>



int factorial(int n);

void main()
 {
   int num, ans;

   printf("Enter Any Number: ");
   scanf("%d", &num);


   ans = factorial(num);

   printf("\nFactorial : %d", ans);


}


int factorial(int n)
{
```

```
    int ans = 1, i = n;

    while (i > 0)
    {
        ans = ans * i;
        i--;
    }
    return ans;
}
```

9. Arrays in C

THEORY EXERCISE:

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

Arrays in C are a collection of elements of the same data type stored in contiguous memory locations. They allow you to store multiple values under a single variable name and access them using an index. Arrays are useful for organizing and managing large sets of data efficiently.

- **Concept of Arrays in C**

➢ Declaration: You specify the type of elements and the number of elements the array will hold.

➢ Indexing: Arrays use zero-based indexing, meaning the first element is accessed with index 0.

➢ Fixed Size: The size of an array must be defined at compile time (except when using dynamic memory allocation).

➢ Homogeneous Elements: All elements in the array have the same data type.

- <u>One-Dimensional Arrays:</u>

A one-dimensional array is a linear sequence of elements. It looks like a list of values.

Example:

```
int numbers[5];

numbers = 10;
numbers = 20;
numbers = 30;
numbers = 40;
numbers = 50;

printf("%d", numbers);

 // Output: 30
```

Here, numbers is a one-dimensional array with 5 elements indexed from 0 to 4.

- Multi-Dimensional Arrays:

Multi-dimensional arrays are arrays of arrays. The most common type is the two-dimensional array, which can be thought of as a table or matrix with rows and columns.

Example of a Two-Dimensional Array:

```
int matrix;
```

```
matrix = 1;
matrix = 2;
matrix = 3;
matrix = 12;

printf("%d", matrix);
 // Output: 3
```

This matrix has 3 rows and 4 columns. You access elements using two indices: the row index and the column index.

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

Aspect	One-Dimensional Array	Multi-Dimensional Array
Structure	Linear sequence (single list)	Array of arrays (table-like structure)
Indices	Single index	Multiple indices (e.g., row and column)
Declaration example	int arr;	int mat;
Use case	Simple lists of elements	Matrices, grids, or higher-dimensional data
Memory layout	Contiguous block of elements	Contiguous block but accessed via multiple indices
Access syntax	arr[i]	mat[i][j]

In summary, arrays in C are fixed-size, same-type collections indexed from zero. One-dimensional arrays are simple lists, while multi-dimensional arrays are more like grids or tables with multiple indices for access. This difference is very useful when dealing with complex data structures like matrices or multi-layered data.

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.

```c
#include <stdio.h>

int main()
{

   int arr[5];
   int i;

   printf("\nEnter 5 integers : ");
   for (i = 0; i < 5; i++)
    {
      scanf("%d", &arr[i]);
    }

   printf("\nYou entered (1D Array) : ");
   for (i = 0; i < 5; i++)
    {
      printf("%d ", arr[i]);
    }
    printf("\n");
```

```
  int matrix[3][3];
  int sum = 0, r, c;

  printf("\nEnter values : ");

  for (r= 0; r < 3; r++)
  {
     for (c= 0; c< 3; c++)
     {
        scanf("%d", &matrix[r][c]);
        sum += matrix[r][c];
     }
  }

  printf("\nur matrix : ");
  for (r = 0; r < 3; r++)
  {
     for (c = 0; c < 3; c++)
     {
        printf(" %d ", matrix[r][c]);
     }
     printf("\n");
  }

  printf("\nSum of all numbers in matrix is = %d ", 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?

- What are Pointers in C?

➢ Pointers in C are special variables that store the memory address of another variable, rather than its direct value. This means that a pointer “points” to the location in memory where data is stored. For example, while an int variable might store the value 10, an int pointer stores the address of where that integer is kept.

- Declaration and Initialization of Pointers:

➢ Declaration-

A pointer is declared by specifying the data type it will point to, followed by an asterisk (*), and then the pointer’s name.

Example:

```
int *ptr;      // pointer to an integer
char *ch_ptr;   // pointer to a character
float *f_ptr;   // pointer to a float
```

➢ Initialization-

To initialize a pointer, assign it the memory address of a variable using the address-of operator &.

Example:

```
int num = 25;
int *ptr = &num;    // ptr now holds the address of num
```

You can also assign NULL to a pointer if it doesn't point to any variable yet:

```
int *ptr = NULL;    // safe initialization with no address
```

- Why are Pointers Important in C?

➢ Direct Memory Access: Pointers allow you to access and modify memory locations directly, providing more control over how data is stored and managed.

➢ Efficient Data Handling: Arrays, strings, and complex data structures like linked lists rely on pointers for efficient manipulation and access.

- Dynamic Memory Allocation: Functions like malloc() and free() use pointers to allocate and release memory at runtime.

- Function Arguments: Pointers enable functions to modify the values of variables passed to them, making it possible to use pass-by-reference.

- Building Data Structures: Advanced structures such as linked lists, trees, and graphs are constructed using pointers to create dynamic relationships between elements.

LAB EXERCISE:

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

```c
#include <stdio.h>

int main()
{
   int a = 11;
   int *ptr = &a;

   printf("\nBefore modification num = %d ", a);

   *ptr = 4;

   printf("\nAfter modification num = %d ", a);

   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.

➢ strings are arrays of characters terminated by a null character ('\0'). Several standard library functions help manipulate and work with strings efficiently

- strlen():

➢ Purpose: Returns the length of a string, excluding the terminating null character.

➢ Usefulness: When you need to know how many characters a string contains.

Example:
If you want to print the length of a user's name or check if a string is empty.

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

int main()
```

```
{
    char ch[] = "android";
    printf("%d",strlen(ch));

   return 0;
}
```

//output=7

- strcpy():

➢ Purpose: Copies one string into another.

➢ Usefulness: To assign or duplicate string content.

Example:
Copying a source string into a destination buffer.

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

int main()
{
   char ch[50];
   char ch2[50];

   printf("Enter Your Name: ");
   scanf("%s",ch);

   strcpy(ch2,ch);
```

```
   printf("Your Name is : %s",ch2);


   return 0;
}
```

//output=
Enter Your Name: abhay
Your Name is : abhay

- strcat():

➢ Purpose: appends one string to the end of another.

➢ Usefulness: To build or combine strings dynamically.

Example:

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

int main()
{
   char fname[]="abhay";
   char lname[]="chandvaniya";

   printf("%s",strcat(fname,lname));



   return 0;
}
```

//output=

abhaychandvaniya

- strcmp():

➢ Purpose: Compares two strings lexicographically.

➢ Usefulness: To check if strings are equal or to sort strings alphabetically.

Example:

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

int main()
{
   char ch[]="mango";
   char ch2[50];

   do
   {
     printf("Fav. Fruit?");
     scanf("%s",ch2);
   }
   while(strcmp(ch,ch2)!=0);
```

```
   printf("Answer is Correct");


   return 0;
}
```

//output=

Fav. Fruit?mango

Answer is Correct

- strchr():

➢ Purpose: Finds the first occurrence of a character in a string.

➢ Usefulness: To locate a character inside a string.

➢ e.g., finding '@' in an email address.

Example:
Searching for a character in a string.

```
char email[] = "user@example.com";
char *ptr = strchr(email, '@');
if(ptr != NULL)
 {
   printf("Found '@' at position: %ld\n", ptr - email);
}
else
 {
   printf("'@' not found\n");
}
```

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()

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

int main()
{
   char str1[10], str2[10];


   printf("Enter the first string: ");
   scanf("%s",str1);

   printf("Enter the second string: ");
   scanf("%s", str2);


   strcat(str1, str2);


   printf("\nConcatenated String: %s", str1);

   printf("\nLength of concatenated string: %d", 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.

- Structures in C:

➢ A structure in C is a user-defined data type that allows grouping variables of different types under a single name. It helps to organize related data together, making it easier to manage and use throughout the program.

- How to Declare a Structure?

➢ To declare a structure, use the struct keyword followed by the structure name and a block containing its members (variables).

Syntax:

```
struct StructureName
{
   dataType member1;
   dataType member2;
   // more members if needed
};
```

Example:

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

- How to Create and Initialize Structure Variables?

After declaring the structure, you can create variables of that type.

➢ Declaration:

```
struct Student s1;
```

➢ Initialization (after declaration):

```
strcpy(s1.name, "Alice");
s1.age = 20;
s1.marks = 85.5;
```

➢ Initialization while declaring variable:

```
struct Student s2 = {"Bob", 22, 90.0};
```

- How to Access Structure Members?

➢ Use the dot operator (.) to access members of a structure variable.

```
printf("Name: %s\n", s1.name);
```

```
printf("Age: %d\n", s1.age);
printf("Marks: %.2f\n", s1.marks);
```

➢ If you have a pointer to a structure, use the arrow operator (->) to access its members:

```
struct Student *ptr = &s1;
printf("Name: %s\n", ptr->name);
ptr->marks = 95.0;
```

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.

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

struct Student {
   char name[50];
   int roll;
   float marks;
}s[3];

int main()
{

   int i;


   for (i = 0; i < 3; i++)
   {
      printf("\nEnter Name : ");
      scanf("%s", s[i].name);

      printf("\nEnter Roll Number : ");
      scanf("%d", &s[i].roll);
```

```
    printf("\nEnter Marks : ");
    scanf("%f", &s[i].marks);
  }



  for (i = 0; i < 3; i++)
  {
    printf("\nName : %s ", s[i].name);
    printf("\nRoll No : %d", s[i].roll);
    printf("\nMarks : %f", s[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.

- Importance of File Handling in C:

➢ File handling in C is important because it allows a program to store data permanently by saving it to files. Unlike variables stored in memory, which lose their values after the program ends, files keep data safe on the disk.

➢ This helps in:

1. Saving and retrieving data between program runs.
2. Sharing data between different programs.
3. Handling large amounts of data beyond memory limits.
4. Reading data from input files and writing output files such as logs or reports.

- File Operations in C:

➢ Opening a File-

To open a file, use the fopen() function with the file name and mode (read, write, append):

```
// open file for reading
FILE *fp = fopen("filename.txt", "r");
```

Common modes include:

Mode	Purpose
"r"	Open file for reading
"w"	Open file for writing (creates or overwrites)
"a"	Open file for appending
"r+"	Open for reading and writing
"w+"	Open for writing and reading (overwrite)
"a+"	Open for appending and reading

If fopen() fails, it returns NULL.

➢ Closing a File-

-After file operations, close the file with:

fclose(fp);

➢ Reading from a File-

1. fgetc(fp) — reads a single character.
2. fgets(buffer, size, fp) — reads a line of text into a buffer.
3. fread(ptr, size, count, fp) — reads binary data.

➢ Writing to a File-

1. fputc(ch, fp) — writes a character.
2. fputs(str, fp) — writes a string.
3. fprintf(fp, "format", ...) — formatted write.
4. fwrite(ptr, size, count, fp) — writes binary data.

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.

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

int main()
{
   FILE *h1;
   char details[]="yooo";
   char c[100];                //bcz we need to read from file

   h1 = fopen("main file.text","w");
   fprintf(h1,"%s",details);

   fclose(h1);

   h1 = fopen("main file.text","r");
   fgets(c , 100 , h1);    //fgets= for reading string from file
   fclose(h1);

   printf("%s",c);

   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 user and perform the respective perform the respective operation (addition, subtraction, multiplication, division or modulus) using operators.

Challenge: Extend the program to handle invalid operator inputs.

```c
#include <stdio.h>

int main()
{
   double num1, num2;
   char operator;

   printf("Enter first number: ");
   scanf("%lf", &num1);

   printf("Enter an operator (+, -, *, /, %%): ");
   scanf(" %c", &operator);  // space before %c to consume any leftover whitespace

   printf("Enter second number: ");
   scanf("%lf", &num2);

   switch (operator)
```

```
  {
    case '+':
      printf("Result: %.2lf\n", num1 + num2);
      break;
    case '-':
      printf("Result: %.2lf\n", num1 - num2);
      break;
    case '*':
      printf("Result: %.2lf\n", num1 * num2);
      break;
    case '/':
      if (num2 != 0)
        printf("Result: %.2lf\n", num1 / num2);
      else
        printf("Error: Division by zero is not allowed.\n");
      break;
    case '%':
      // modulus only works with integers
      if ((int)num2 != 0)
        printf("Result: %d\n", (int)num1 % (int)num2);
      else
        printf("Error: Modulus by zero is not allowed.\n");
      break;
    default:
      printf("Error: Invalid operator '%c'. Please use one of +, -, *, /, %%.\n",
operator);
  }

  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:

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

```c
#include <stdio.h>

int main()
{
    int number;

    // Get input from the user
    printf("Enter an integer: ");
    scanf("%d", &number);

    // Check even or odd
    if (number % 2 == 0)
        printf("The number is even.\n");
    else
        printf("The number is odd.\n");

    // Check positive, negative, or zero
    if (number > 0)
        printf("The number is positive.\n");
    else if (number < 0)
        printf("The number is negative.\n");
    else
        printf("The number is zero.\n");

    // Check if multiple of both 3 and 5
    if (number % 3 == 0 && number % 5 == 0)
        printf("The number is a multiple of both 3 and 5.\n");
```

```
    else
        printf("The number is 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 student as input and displays the corresponding grade base 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.

```c
#include <stdio.h>

int main()
{
    float marks;
    int gradeCategory;

    // Input marks
    printf("Enter student marks (0 to 100): ");
    scanf("%f", &marks);

    // Determine grade category using if-else
    if (marks < 0 || marks > 100)
    {
        printf("Invalid marks! Please enter a value between 0 and 100.\n");
        return 1; // exit the program
```

```
    }
    else if (marks > 90)
    {
        gradeCategory = 1;  // Grade A
    }
    else if (marks > 75)
    {
        gradeCategory = 2;  // Grade B
    }
    else if (marks > 50)
    {
        gradeCategory = 3;  // Grade C
    }
    else
    {
        gradeCategory = 4;  // Grade D
    }

    // Use switch to display the grade
    switch (gradeCategory)
    {
        case 1:
            printf("Grade A\n");
            break;
        case 2:
            printf("Grade B\n");
            break;
        case 3:
            printf("Grade C\n");
            break;
        case 4:
            printf("Grade D\n");
            break;
        default:
            printf("Unknown grade category.\n");
    }

    return 0;
}
```

- LAB EXERCISE 2: Number Comparison

Write a C program that takes three number from the user and determines:

- The largest number.
- The smallest number.

Challenge: Solve the problem using both if-else and switch-case statements.

```c
#include <stdio.h>

int main()
{
   int num1, num2, num3;
   int largestCode, smallestCode;

   // Input three numbers
   printf("Enter three numbers:\n");
   scanf("%d %d %d", &num1, &num2, &num3);

   // Determine the largest number

   if (num1 >= num2 && num1 >= num3)
      largestCode = 1;
   else if (num2 >= num1 && num2 >= num3)
      largestCode = 2;
   else
      largestCode = 3;
   // Determine the smallest number

   if (num1 <= num2 && num1 <= num3)
      smallestCode = 1;
   else if (num2 <= num1 && num2 <= num3)
      smallestCode = 2;
   else
```

```
    smallestCode = 3;

  // Output the largest using switch-case
  switch (largestCode)
  {
    case 1:
      printf("The largest number is: %d\n", num1);
      break;
    case 2:
      printf("The largest number is: %d\n", num2);
      break;
    case 3:
      printf("The largest number is: %d\n", num3);
      break;
    default:
      printf("Error determining the largest number.\n");
  }

  // Output the smallest using switch-case
  switch (smallestCode)
  {
    case 1:
      printf("The smallest number is: %d\n", num1);
      break;
    case 2:
      printf("The smallest number is: %d\n", num2);
      break;
    case 3:
      printf("The smallest number is: %d\n", num3);
      break;
    default:
      printf("Error determining the smallest number.\n");
  }

  return 0;
}
```

- LAB EXERCISE 3: Temperature calculation

Write a C program to read temperature in centigrade and display a suitable message according to the temperature state below: Temp < 0 then freezing weather temp 0-10 then vary cold weather Temp 10-20 then Cold weather Temp 20-30 then Normal in Temp Temp 30-40 then its Hot Temp >=40 then its very hot

```c
#include <stdio.h>

int main()
{
   float temp;

   // Input temperature
   printf("Enter temperature in centigrade: ");
   scanf("%f", &temp);

   // Determine weather condition
   if (temp < 0)
   {
      printf("Freezing weather\n");
   }
   else if (temp >= 0 && temp < 10)
   {
      printf("Very Cold weather\n");
   }
   else if (temp >= 10 && temp < 20)
   {
      printf("Cold weather\n");
   }
   else if (temp >= 20 && temp < 30)
   {
```

```
        printf("Normal in Temp\n");
    }
    else if (temp >= 30 && temp < 40)
    {
        printf("It's Hot\n");
    }
    else if (temp >= 40)
    {
        printf("It's Very Hot\n");
    }
    else
    {
        printf("Invalid temperature input.\n");
    }

    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.

```c
#include<stdio.h>

int main()
{
   int n, i, j, isPrime;

   printf("Enter the number limit: ");
   scanf("%d", &n);

   printf("Prime numbers between 1 and %d are:\n", n);

   for (i = 2; i <= n; i++)
   {
      isPrime = 1;
      for (j = 2; j <= i / 2; j++)
      {
         if (i % j == 0)
         {
            isPrime = 0;
            break;
         }
      }
```

```
        if (isPrime)
            printf("%d ", i);
    }

    printf("\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).

```c
#include<stdio.h>

int main()
{
   int num, start, end, i;

   printf("Enter a number to print its multiplication table: ");
   scanf("%d", &num);

   printf("Enter the start of the range: ");
   scanf("%d", &start);

   printf("Enter the end of the range: ");
   scanf("%d", &end);

   // Handle if start > end
   if (start > end)
   {
      printf("Invalid range! Start should be less than or equal to end.\n");
      return 1;
   }

   printf("Multiplication table of %d from %d to %d:\n", num, start, end);
   for (i = start; i <= end; i++)
   {
      printf("%d x %d = %d\n", num, i, num * 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.

```c
#include<stdio.h>

int main()
{
   int num, sum = 0, reverse = 0, digit;

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

   int temp = num; // Store original number

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

   printf("Sum of digits of %d is %d\n", temp, sum);
   printf("Reversed number is %d\n", reverse);

   return 0;
}
```

- **Patterns**

1)

```
#include<stdio.h>

void main()
{

   int i,j;

   for(i = 1; i <= 5; i++)
   {
      for(j = 1; j <= i; j++)
      {
         printf("%d ", j % 2);
      }
      printf("\n");

   }
   getch();
}
```

2)

```c
#include<stdio.h>

void main()
{
   int i, j;
   char ch = 'A';

   for(i = 1; i <= 5; i++)
   {
      for(j = 1; j <= i; j++)
      {
         printf("%c ", ch);
         ch++; // Move to next alphabet
      }
      printf("\n");
   }
   getch(); // Optional on modern systems
}
```

3)

```c
#include<stdio.h>

void main()
{
   int i, j;
   int n = 1;

   for(i = 1; i <= 5; i++)
   {
      for(j = 1; j <= i; j++)
      {
         printf("%d ", n);
         n++; // Move to next alphabet
      }
      printf("\n");
   }
   getch(); // Optional on modern systems
}
```

4)

```c
#include<stdio.h>

void main()
{
   int i, j;
   char ch;

   for(i = 1; i <= 5; i++)
   {
      ch = 'A';
      for(j = 1; j <= i; j++)
      {
         printf("%c ", ch);
         ch++; // Move to next alphabet
      }
      printf("\n");
   }
   getch(); // Optional on modern systems
}
```

5)

```c
#include <stdio.h>

int main()
{
    int i, j, space;

    for (i = 1; i <= 5; i++)
    {
        // Print leading spaces
        for (space = 1; space <= 5 - i; space++)
        {
            printf("  ");  // Two spaces for alignment
        }

        // Print stars with space
        for (j = 1; j <= (2 * i - 1); j++)
        {
            printf("* ");
        }

        // Newline after each row
        printf("\n");
    }

    return 0;
}
```

6)

```c
#include<stdio.h>

int main()
{
   int i, j;

   // Top half
   for (i = 1; i <= 6; i++)
   {

      // Print stars
      for (j = 1; j <= i; j++)
      {
         printf("* ");
      }
      printf("\n");
   }

   // Bottom half
   for (i = 5; i >= 1; i--)
   {
      // Print stars
      for (j = 1; j <= i; j++)
      {
         printf("* ");
      }
      printf("\n");
   }

   return 0;
}
```

Write a program to find out the max from given number (E.g., No: -1562 Max number is 6)

```c
#include<stdio.h>

int main()
{
   int num, digit, max = 0;

   printf("Enter a number: ");
   scanf("%d", &num);

   // If number is negative, make it positive
   if (num < 0)
   {
      num = num;
   }

   // Loop through digits
   while (num > 0)
   {
      digit = num % 10;     // Get last digit
      if (digit > max)
      {
         max = digit;      // Update max if needed
      }
      num = num / 10;       // Remove last digit
   }
   printf("Maximum digit is: %d\n", max);
   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.

```c
#include<stdio.h>

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

   // Input: Accept 10 integers
   printf("Enter 10 integers:\n");
   for (i = 0; i < 10; i++)
   {
      printf("Enter number %d: ", i + 1);
      scanf("%d", &arr[i]);
   }

   // Initialize max and min
   max = min = arr[0];

   // Find max and min
   for (i = 1; i < 10; i++)
   {
```

```
        if (arr[i] > max)
        {
            max = arr[i];
        }
        if (arr[i] < min)
        {
            min = arr[i];
        }
    }

    // Output max and min
    printf("\nMaximum value: %d\n", max);
    printf("Minimum value: %d\n", min);

    // Challenge: Sort array in ascending order (using simple bubble sort)
    for (i = 0; i < 9; i++)
    {
        for (int j = i + 1; j < 10; j++)
        {
            if (arr[i] > arr[j])
            {
                // Swap arr[i] and arr[j]
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }

    // Output sorted array
    printf("\nArray in ascending order:\n");
    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }

    printf("\n");
    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.

```c
#include <stdio.h>

int main()
{
   int a[2][2], b[2][2], sum[2][2];
   int i, j;

   // Input first matrix
   printf("Enter elements of first 2x2 matrix:\n");
   for (i = 0; i < 2; i++)
{
      for (j = 0; j < 2; j++)
{
         scanf("%d", &a[i][j]);
      }
   }

   // Input second matrix
   printf("Enter elements of second 2x2 matrix:\n");
   for (i = 0; i < 2; i++)
{
      for (j = 0; j < 2; j++)
{
         scanf("%d", &b[i][j]);
      }
   }

   // Matrix addition
```

```
    for (i = 0; i < 2; i++)
{
        for (j = 0; j < 2; j++)
{
            sum[i][j] = a[i][j] + b[i][j];
        }
    }

    // Display result
    printf("Result matrix after addition:\n");
    for (i = 0; i < 2; i++)
{
        for (j = 0; j < 2; j++)
{
            printf("%d\t", sum[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

```c
#include<stdio.h>

int main()
{

   int c[3][3],d[3][3],result[3][3]={0};
   int y,z,a;

   printf("Enter 1st matrix of 3*3: ");

   for(y=0;y<3;y++)
   {
      for(z=0;z<3;z++)
      {
         scanf("%d",&c[y][z]);
      }
   }

   printf("Enter 2nd matrix of 3*3: ");

   for(y=0;y<3;y++)
   {
      for(z=0;z<3;z++)
      {
         scanf("%d",&d[y][z]);
      }
   }

   for(y=0;y<3;y++)//row
   {
      for(z=0;z<3;z++)//col
      {
         for(a=0;a<3;a++)
         {
            result[y][z]+= c[y][a]*d[a][z];

         }
      }
   }
```

```
   printf("Result matrix is: ");
   for(y=0;y<3;y++)
   {
      for(z=0;z<3;z++)
      {
        printf("%d ",result[y][z]);
      }
      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.

```c
#include <stdio.h>

int main()
{
   int N, i;
   float sum = 0, average;

   // Ask the user for the number of elements
   printf("Enter the number of elements: ");
   scanf("%d", &N);

   // Declare an array of size N
   float numbers[N];

   // Get the numbers from the user
   printf("Enter %d numbers:\n", N);
   for(i = 0; i < N; i++)
   {
      printf("Number %d: ", i + 1);
      scanf("%f", &numbers[i]);
      sum += numbers[i];  // Add to sum while reading
   }

   // Calculate average
   average = sum / N;
```

```
    // Display the results
    printf("\nSum of the numbers: %.2f\n", sum);
    printf("Average of the numbers: %.2f\n", average);

    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.

```c
#include<stdio.h>

// Recursive function to return nth Fibonacci number
int fibonacci(int n)
{
   if (n == 0)
      return 0;
   else if (n == 1)
      return 1;
   else
      return fibonacci(n - 1) + fibonacci(n - 2);
}

int main()
{
   int n, i;

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

   printf("Fibonacci Sequence up to %d terms:\n", n);
   for (i = 0; i < n; i++)
   {
      printf("%d ", fibonacci(i));
   }

   return 0;
}
```

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

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

// Recursive Fibonacci (inefficient)
int fibonacci_recursive(int n)
{
    if (n <= 1)
        return n;
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2);
}

// Iterative Fibonacci (efficient)
int fibonacci_iterative(int n)
{
    if (n <= 1)
        return n;

    int a = 0, b = 1, temp;
    for (int i = 2; i <= n; i++)
    {
        temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}

int main()
{
    int n;
    printf("Enter the value of n: ");
    scanf("%d", &n);

    // Timing recursive version
    clock_t start_recursive = clock();
```

```c
    int result_recursive = fibonacci_recursive(n);
    clock_t end_recursive = clock();
    double time_recursive = (double)(end_recursive - start_recursive) /
CLOCKS_PER_SEC;

    // Timing iterative version
    clock_t start_iterative = clock();
    int result_iterative = fibonacci_iterative(n);
    clock_t end_iterative = clock();
    double time_iterative = (double)(end_iterative - start_iterative) /
CLOCKS_PER_SEC;

    // Output results
    printf("\nFibonacci number (Recursive) for %d\n", result_recursive);
    printf("Time taken (Recursive): %f seconds\n", time_recursive);

    printf("\nFibonacci number (Iterative) for %d\n", result_iterative);
    printf("Time taken (Iterative): %f seconds\n", time_iterative);

    return 0;
}
```

- LAB EXERCISE 2: Factorial Calculation

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

```c
#include<stdio.h>

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

int main()
{
   int num;

   printf("Enter a number: ");
   scanf("%d", &num);

   if (num < 0)
   {
      printf("Factorial is not defined for negative numbers.\n");
   }
   else
   {
      int fact = factorial(num);
      printf("Factorial of %d is: %d\n", num, fact);
   }

   return 0;
}
```

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

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

// Recursive factorial function
double factorial_recursive(int n)
{
   if (n <= 1)
      return 1;
   return n * factorial_recursive(n - 1);
}

// Iterative factorial function
double factorial_iterative(int n)
{
   double result = 1;
   for (int i = 1; i <= n; i++)
   {
      result *= i;
   }
   return result;
}

int main()
{
   int num;
   clock_t start, end;
   double time_recursive, time_iterative;

   printf("Enter a number: ");
   scanf("%d", &num);

   if (num < 0)
   {
      printf("Factorial is not defined for negative numbers.\n");
      return 0;
```

```
    }

    // Recursive
    start = clock();
    double rec_result = factorial_recursive(num);
    end = clock();
    time_recursive = (double)(end - start) / CLOCKS_PER_SEC;

    // Iterative
    start = clock();
    double itr_result = factorial_iterative(num);
    end = clock();
    time_iterative = (double)(end - start) / CLOCKS_PER_SEC;

    // Output results
    printf("\n--- Factorial Results ---\n");
    printf("Recursive Result: %lf\n", rec_result);
    printf("Time (recursive): %.6f sec\n", time_recursive);

    printf("\nIterative Result: %lf\n", itr_result);
    printf("Time (iterative): %.6f sec\n", time_iterative);

    return 0;
}
```

- 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.

```c
#include<stdio.h>

// Function to check if a number is a palindrome
int isPalindrome(int num)
{
    int original = num;
    int reversed = 0;

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

    return (original == reversed);
}

int main()
{
    int number;

    printf("Enter a number: ");
    scanf("%d", &number);

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

    return 0;
}
```

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

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

// Function to check if a string is a palindrome
int isPalindrome(char str[])
{
    int start = 0;
    int end = strlen(str) - 1;

    while (start < end)
    {
        // Convert both to lowercase for case-insensitive comparison
        if (tolower(str[start]) != tolower(str[end]))
            return 0;  // Not a palindrome
        start++;
        end--;
    }
    return 1;  // Palindrome
}

int main()
{
    char str[100];

    printf("Enter a string: ");
    scanf("%s", str);  // For simple words (no spaces)

    if (isPalindrome(str))
        printf("\"%s\" is a palindrome.\n", str);
    else
        printf("\"%s\" is not a palindrome.\n", str);

    return 0;
}
```

6. Strings

- LAB EXERCISE 1: String Reversal

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

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

// Function to reverse the string
void reverseString(char str[])
{
   int len = strlen(str);
   int i;
   char temp;

   for (i = 0; i < len / 2; i++)
   {
      // Swap characters
      temp = str[i];
      str[i] = str[len - i - 1];
      str[len - i - 1] = temp;
   }
}

int main()
{
   char str[100];

   printf("Enter a word: ");
   scanf("%s", str);  // Reads string without spaces

   reverseString(str);

   printf("Reversed word: %s\n", str);
```

```
    return 0;
}
```

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

```c
#include<stdio.h>

// Function to calculate length of string manually
int stringLength(char str[])
{
   int length = 0;
   while (str[length] != '\0')
   {
      length++;
   }
   return length;
}

// Function to reverse the string
void reverseString(char str[])
{
   int len = stringLength(str);
   int i;
   char temp;

   for (i = 0; i < len / 2; i++)
   {
      // Swap characters
      temp = str[i];
      str[i] = str[len - i - 1];
      str[len - i - 1] = temp;
   }
}

int main()
{
   char str[100];

   printf("Enter a word: ");
   scanf("%s", str);  // Reads one word (no spaces)
```

```
    reverseString(str);

    printf("Reversed word: %s\n", str);

    return 0;
}
```

- 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.

```c
#include <stdio.h>

// Function to check if a character is a vowel (manual lowercase check)
int isVowel(char ch)
{
    if (ch >= 'A' && ch <= 'Z')
    {
        ch = ch + 32; // Convert to lowercase
    }
    return (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u');
}

// Function to check if a character is an alphabet
int isAlphabet(char ch)
{
    return ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z'));
}

int main()
{
    char str[100];
    int vowels = 0, consonants = 0;
    int i = 0;
    char ch;

    printf("Enter a string: ");

    // Manual character input (until newline or max size)
```

```
    while (i < 99)
    {
        ch = getchar();
        if (ch == '\n')
        {
            break;
        }
        str[i] = ch;

        if (isAlphabet(ch))
        {
            if (isVowel(ch))
            {
                vowels++;
            }
            else
            {
                consonants++;
            }
        }

        i++;
    }

    str[i] = '\0'; // Null-terminate the string manually

    // Output
    printf("Number of vowels: %d\n", vowels);
    printf("Number of consonants: %d\n", consonants);

    return 0;
}
```

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

```c
#include <stdio.h>

// Function to check if a character is a vowel (manual lowercase check)
int isVowel(char ch) {
    if (ch >= 'A' && ch <= 'Z') {
        ch = ch + 32; // Convert to lowercase
    }
    return (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u');
}

// Function to check if a character is an alphabet
int isAlphabet(char ch) {
    return ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z'));
}

int main() {
    char str[100];
    int vowels = 0, consonants = 0;
    int i = 0;
    char ch;

    printf("Enter a string: ");

    // Manual character input (until newline or max size)
    while (i < 99) {
        ch = getchar();
        if (ch == '\n') {
            break;
        }
        str[i] = ch;

        if (isAlphabet(ch)) {
            if (isVowel(ch)) {
                vowels++;
            } else {
                consonants++;
```

```
        }
      }

      i++;
    }

    str[i] = '\0'; // Null-terminate the string manually

    // Output
    printf("Number of vowels: %d\n", vowels);
    printf("Number of consonants: %d\n", consonants);

    return 0;
}
```

- LAB EXERCISE 3: Word Count

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

```c
#include <stdio.h>

int main()
{
   char ch;
   int inWord = 0;     // Flag to track if we're inside a word
   int wordCount = 0;

   printf("Enter a sentence: ");

   while ((ch = getchar()) != '\n')
   {
      // If the character is a space or tab, we're between words
      if (ch == ' ' || ch == '\t')
      {
         inWord = 0;
      }
      // If the character is not a space and we're not already in a word
      else if (inWord == 0)
      {
         inWord = 1;
         wordCount++; // We just found a new word
      }
   }

   printf("Number of words: %d\n", wordCount);

   return 0;
}
```

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

```c
#include <stdio.h>

int main()
{
   char ch;
   char longestWord[100];
   char currentWord[100];
   int wordCount = 0;
   int maxLen = 0;
   int currLen = 0;
   int inWord = 0;
   int i = 0;

   printf("Enter a sentence: ");

   while ((ch = getchar()) != '\n')
   {
      if (ch == ' ' || ch == '\t')
      {
         if (inWord)
         {
            currentWord[currLen] = '\0'; // End current word
            wordCount++;

            if (currLen > maxLen)
            {
               maxLen = currLen;

               // Copy currentWord to longestWord manually
               int j;
               for (j = 0; j <= currLen; j++)
               {
                  longestWord[j] = currentWord[j];
               }
            }
```

```
            currLen = 0;  // Reset length for next word
            inWord = 0;   // We're outside a word now
        }
    }
    else
    {
        if (!inWord)
        {
            inWord = 1;
            currLen = 0; // Start of new word
        }

        currentWord[currLen] = ch;
        currLen++;
    }
}

// Handle the last word if sentence didn't end with a space
if (inWord)
{
    currentWord[currLen] = '\0';
    wordCount++;

    if (currLen > maxLen)
    {
        maxLen = currLen;
        int j;
        for (j = 0; j <= currLen; j++)
        {
            longestWord[j] = currentWord[j];
        }
    }
}

// Output the results
printf("\nNumber of words: %d\n", wordCount);
printf("Longest word: %s\n", longestWord);

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).

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

int main()
{
   int num, originalNum, remainder, result = 0, n = 0;

   // Input a number
   printf("Enter an integer: ");
   scanf("%d", &num);

   originalNum = num;

   // Count the number of digits
   while (originalNum != 0)
   {
      originalNum /= 10;
      n++;
   }

   originalNum = num;

   // Calculate the sum of nth power of its digits
   while (originalNum != 0)
   {
      remainder = originalNum % 10;
      result += pow(remainder, n);
      originalNum /= 10;
   }

   // Check if the result equals the original number
```

```
    if (result == num)
        printf("%d is an Armstrong number.\n", num);
    else
        printf("%d is not an Armstrong number.\n", num);

    return 0;
}
```

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

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

int main()
{
    int num, originalNum, remainder, result, n;

    printf("Armstrong numbers between 1 and 1000 are:\n");

    for (num = 1; num <= 1000; num++)
    {
        originalNum = num;
        result = 0;

        // Count digits
        n = 0;
        int temp = originalNum;
        while (temp != 0)
        {
            temp /= 10;
            n++;
        }

        temp = originalNum;

        // Calculate sum of digits raised to the power n
        while (temp != 0)
        {
            remainder = temp % 10;
            result += pow(remainder, n);
            temp /= 10;
        }

        if (result == num)
        {
            printf("%d\n", num);
```

```
    }
  }

  return 0;
}
```

- Lab Challenge 2: Pascal's Triangle

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

```c
#include<stdio.h>

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

// Function to calculate nCr (combinations)
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);

   // Generate Pascal's Triangle
   for (int i = 0; i < rows; i++)
   {
      // Print spaces for formatting
      for (int space = 0; space < rows - i - 1; space++)
      {
         printf("  ");
```

```
        }

        for (int j = 0; j <= i; j++)
        {
            printf("%4d", combination(i, j));  // Print nCr
        }

        printf("\n");
    }

    return 0;
}
```

Challenge: Implement the same program using a recursive function.

```c
#include<stdio.h>

// Recursive function to calculate nCr
int combination(int n, int r)
{
    if (r == 0 || r == n)
        return 1;
    else
        return combination(n - 1, r - 1) + combination(n - 1, r);
}

int main()
{
    int rows;

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

    // Generate Pascal's Triangle
    for (int i = 0; i < rows; i++)
    {
        // Print spaces for alignment
        for (int space = 0; space < rows - i - 1; space++)
        {
            printf("  ");
        }
        for (int j = 0; j <= i; j++)
        {
            printf("%4d", combination(i, j));
        }

        printf("\n");
    }

    return 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: Extend the program to find and print all the prime numbers found.

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

int main()
{
   int number, guess, attempts = 10;

   // Seed the random number generator
   number = (rand() % 100) + 1;  // Random number between 1 and 100

   printf("Welcome to the Number Guessing Game!\n");
   printf("Guess the number between 1 and 100.\n");
   printf("You have %d attempts.\n\n", attempts);

   // Game loop
   for (int i = 1; i <= attempts; i++)
   {
      printf("Attempt %d: Enter your guess: ", i);
      scanf("%d", &guess);

      if (guess < number)
      {
         printf("Too low! \n");
      }
      else if (guess > number)
```

```
        {
            printf("Too high! \n");
        }
        else
        {
            printf("Congratulations! ");
            return 0;  // Exit the program
        }
    }

    // If user fails to guess
    printf("you are out of attempts");

    return 0;
}
```

- Lab Challenge 4: Sum of prime numbers

Description: Write a C program that calculates the sum of all prime numbers up to a given number N.

Challenge: Extend the program to find and print all the prime numbers found.

```c
#include<stdio.h>

int main() {
   int N, i, j, isPrime, sum = 0;

   printf("Enter a number N: ");
   scanf("%d", &N);

   printf("Prime numbers up to %d are: ", N);

   for (i = 2; i <= N; i++)
   {
      isPrime = 1; // assume i is prime

      // check if i is divisible by any number from 2 to i-1
      for (j = 2; j < i; j++)
      {
         if (i % j == 0)
         {
            isPrime = 0; // not prime
            break;
         }
      }

      // if prime, print and add to sum
      if (isPrime)
      {
```

```
            printf("%d ", i);
            sum += i;
        }
    }

    printf("\nSum of all prime numbers up to %d is: %d\n", N, sum);

    return 0;
}
```