1. **Overview of C programming**

**->**

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 provides low-level access to memory and allows direct manipulation of hardware, making it ideal for developing operating system, embedded system, and other performance-critical applications.

**THEORY EXERCISE:**

1. **History and Evolution**

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

1. **Importance Why it is used today**

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

**LAB EXERCISE:**

**->**

C programming is extensively used in three major areas: embedded systems and game development. These fields require the precision, performance and low-level control that C provides.

1. **Embedded Systems:**

Embedded systems are specialized computer systems that are embedded within other devices to control specific functions, such as washing machines, microwawe ovens, printers and automobiles. C is favored for embedded systems due to its ability to interact directly with hardware, enabling efficient resources management and low-level control, which are critical for these applications.

1. **Operating System:**

C is the foundation for many operating system, including Windows, Linux and macOS. It’s low-level control and efficient memory management make it ideal for developing the core components of an operating system, such as the kernel, file system, and device drivers.

1. **Game Development:**

C is used in the development of game engineers, particularly for performance -critical section of games. It’s ability to optimize code for speed and its control over hardware resources make it a natural choice for creating efficient game engines.

These three applications demonstrate the versatility and importance of C programming in modern technology, where its ability to interact with hardware and manage resources efficiently remains highly valued.

**(2). Setting up Environment.**

**THEOTY EXERCISE:**

**->**

To install a C compiler, first, download and install a c compiler like **GCC** On Linux, you can install GCC using the command sudoapt-get install gcc. On Windows, you can use **MingGW** 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:**

->

#include<stdio.h>

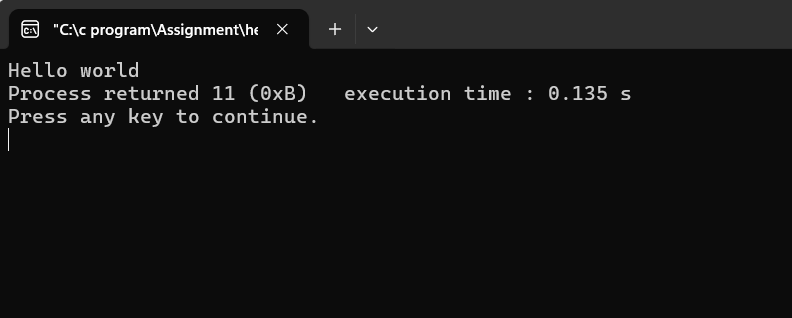
Void main ()

{

Print f (“Hello World”);

}

**Output:**

****

**(3). Basic Structure of C program.**

**->**

A C program is typically consists of three main parts:

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

**THEORY EXERCISE:**

**->**

A C program’s basic structure involves headers, a main function, comments, data types and variables. Header Include libraries for standard functions, while the main function is where program execution begins. Comments explain the code. Data types specify the kind of data stored, and variables are named storage locations for data.

1. **Headers:**

* Headers contain function declarations and other information needed by the compiler.
* They are include using the #include directive.
* For example, #include <Stdio .h> includes the standard input/output library.
* Other common header Include <Stdlib. h> for standard library functions, <String .h> for string manipulation and <math. h> for mathematical functions.
* **Example:**

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

Int main ()

{

//…

}

1. **Main Function:**

* The main () function is where the program starts executing.
* It can have arguments (e.g., int main (int argc, char \*argv [])) for command-line arguments.
* It returns an integer value (typically 0 for success) to the operating system.
* **Example:**

#include <stdio.h>

Int main ()

{// The main function

Print f (“Hello, World! \ n”); // print a message

Return 0; // Return 0 to indicate successful execution

}

1. **Comments:**

* Comments explain the code and are ignored by the compiler.
* Single-line comments start with //.
* Multi-line comments are enclosed in /\*and \*/.
* **Example:**

#include <stdio.h>

Int main ()

{

// This is a single-line comment

/\*

This is a multi-line comment.

It can span multiple lines.

\*/

Print f (“Hello, World! \ n”);

Return 0;

}

1. **Data Types:**

* Data types specify the kind of a variable can store.
* Common data -types include:
* Int: integer number (e, g,.10, -5, 2023)
* Float: floating-point numbers (e, g,.3.14, -2.71)
* Double: double-precision floating-point numbers (e, g,.3.14159)
* Char: single characters (e, g,. ‘a’, ‘B’, ‘!’)
* void: absence of a type (used in function declaration)
* Char and int can be signed or unsigned. Signed by default.
* For example, unsigned int stores only non-negative integer values.
* **Example:**

#include <STDIO .h>

Int main ()

{

Int age = 30; // Integer variable

Float height = 1.75; // Floating-point variable

Char letter = ‘A’;// Character variable

Print f (“Age: %d\n”, age);

Print f (“Height: %f\n”, height);

Print f (“Letter: %c\n”, Letter);

Return 0;

}

1. **Variable:**

* Variables are named storage location in memory.
* They hold data of a specific type.
* Variables must be declared before they can be used.
* **Example:**

#include <STDIO .h>

Int main ()

{

Int age; // Declared an integer variable named age

Age = 30; // Assign a value to the variable age

Print f (“Age: %d\n”, age); // Print the value of the variable age

Return 0;

}

**LAB EXERCISE:**

**->**

#include<stdio.h>

int main ()

{

//constant declaration

const float pi=3.14159;

//variable declaration

int age = 25; // Integer variable

char grade = 'A'; // Character variable

float height = 5.9; // Float variable

// Display the values

Print f ("Constant PI: %.5f\n", pi);

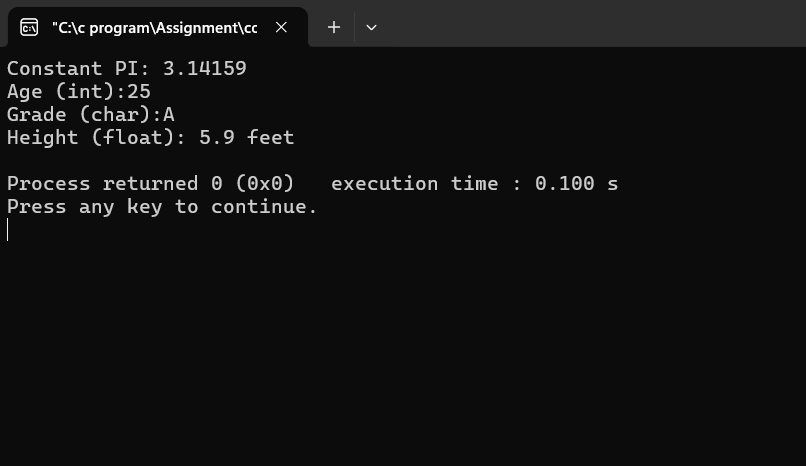
Print f ("Age (int): %d\n", age);

Print f ("Grade (char): %c\n", grade);

Print f ("Height (float): %.1f feet\n", height);

}

**Output:**

****

**(4). Operator in C.**

**->**

A symbol that takes one or more operands such as variables, expression or values and operates on them to give an output.

Ex.- **=, +, -, /, \*, ==, ++, --, %, etc.**

An operator is **a symbol that tells the compiler to perform specific mathematical or logical functions.**

Ex.- **+, -, \*, /, ==, ++, --**

**THEORY EXERCISE:**

**->**

1. **Arithmetic operator.**

**->**

Arithmetic operators in C are symbols that perform mathematical operations on numerical values. These operators are essential for performing calculation within C programs. Here’s a breakdown of the common arithmetic operators.

1. **Relational operator.**

**->**

Relational operator in C Are used to compare two values or expressions. These operators are essential for decision-making in the programs, allowing the program to execute different code blocks based on the comparison results. The result of a relational operation is Boolean value, either true (represented as 1) or false (represented as 0).

1. **Logical operator.**

**->**

In c programming, logical operators are used to combine or negate conditions, which are essential for controlling the flow of a program. The are three main logic operators.

1. **Assignment operator.**

**->**

Assignment operator in C Are used to assign values to variables. The basic assignment operator is the equals sign(=), which assigns the value on the right to the variable on the left.

1. **Increment / Decrement.**

**->**

In C, the increment (++) and decrement (--) operators are unary operators the increase or decrease the value of variable by1. They are commonly used in loops and for updating counters. These operators come in two forms: prefix or postfix.

1. Prefix:

* The operator appears before the variable (e.g.,++a, --a).
* The value of the variable is modified first, and then the modified value is used in the expression.
* Example:

Int a =5;

Int b= ++a; // a become 6, then b is assigned 6

1. Postfix:

* The operator appears after the variable (e.g.,a++, a--).
* The original values of the value of the variable is used in the expression first, and then the value of the variable is modified.
* Example:

Int a=5;

Int b=a++; // b is assigned 5, then a becomes 6

1. **Bitwise operator**

**->**

Bitwise operator programming Manipulate individual bits within a binary representation of numbers. They perform operation like AND, OR, XOR, NOT, left shift, and right shift on the bit level, allowing for efficient low-level data manipulation.

1. **Conditional operator.**

**->**

The conditional operator, also known as the ternary operator, is a shorthand way to write concise if-else-statements in programming. It evaluates a condition and returns one of two possible results based on weather the condition is true or false. The general syntaxis: condition? result \_ if \_true: result \_ if \_false.

**LAB EXERCISE:**

**->**

**#include <STDIO. h>**

**int main ()**

**{**

**int a;**

**int b;**

**// Input two value.**

**Print f ("Enter first value: ");**

**Scan f ("%d", &a);**

**Print f ("Enter second value: ");**

**Scan f ("%d", &b);**

**// Arithmetic Operations**

**Print f ("\n--- Arithmetic Operations ---\n");**

**Print f ("a + b = %d\n", a + b);**

**Print f ("a - b = %d\n", a - b);**

**Print f ("a \* b = %d\n", a \* b);**

**if (b! = 0) {**

**print f ("a / b = %d\n", a / b);**

**print f ("a %% b = %d\n", a % b); // %% to print %**

**}**

**Else**

**{**

**Print f ("Division and modulus not possible (division by zero).\n");**

**}**

**// Relational Operations**

**Print f ("\n--- Relational Operations ---\n");**

**Print f ("a == b: %d\n", a == b);**

**Print f ("a != b: %d\n", a != b);**

**Print f("a > b: %d\n", a > b);**

**Print f ("a < b: %d\n", a < b);**

**Print f ("a >= b: %d\n", a >= b);**

**Print f ("a <= b: %d\n", a <= b);**

**// Logical Operations**

**Print f ("\n--- Logical Operations ---\n");**

**Print f ("a && b: %d\n", a && b); // Logical AND**

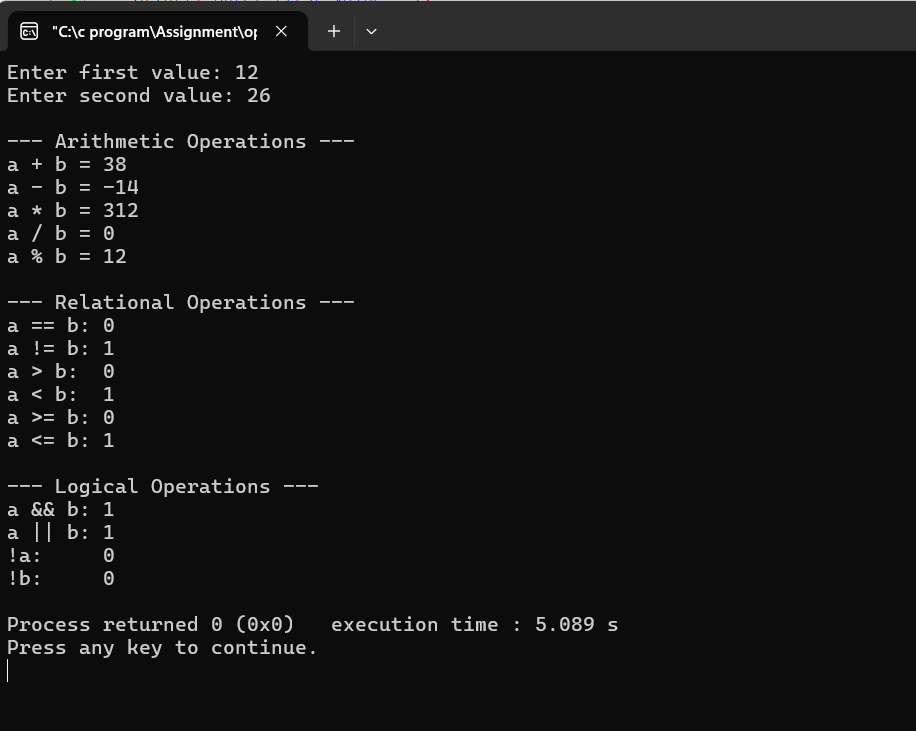
**Print f ("a || b: %d\n", a || b); // Logical OR**

**Print f ("! a: %d\n", !a); // Logical NOT of a**

**Print f ("! b: %d\n", ! b); // Logical NOT of b**

**}**

**Output: -**

****

**(5). Control Flow Statements in C.**

**->**

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

**THEORY EXERCISE:**

**->**

1. **If Statement**

* If statement is the basic decision making statement.
* Used decide whether a certain statement or block of statements will be executed or not.
* **Syntax:**

If (condition)

{

Statement\_1; //true block

Statements

}

Statements x;

1. **If else Statement**

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

If (condition)

{

Statements; //true statement

}

Else

{

Statements; // False statements

}

1. **Nested if Statement**

* Nested if statement is simply an if statements embedded with an another if statement
* **Syntax:**

If (condition1)

{

Statements; // executes when condition1 is true

If (condtition2)

{

Statements; // executes when condition2 is true

}

}

1. **Switch Statement**

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

**:**

Switch (n) //executed when n=1

**{**

Case1: //executed when n=2

Break;

Case2: // executed when n doesn’t match any case

Break;

Default:

**}**

**Lab Exercise:**

**->**

#include <stdio .h>

int main ()

{

int number;

int month;

// Check Even or Odd

Print f ("Enter a number to check even or odd: ");

Scan f ("%d", &number);

if (number % 2 == 0)

{

Print f ("%d is Even\n", number);

}

else

{

Print f ("%d is Odd\n", number);

}

// Month Name using Switch

Prin t f ("Enter a number (1 to 12) to display the month: ");

Scan f ("%d", &month);

switch (month)

{

case 1:

print f ("Month: January\n");

break;

case 2:

print f ("Month: February\n");

break;

case 3:

print f ("Month: March\n");

break;

case 4:

print f ("Month: April\n");

break;

case 5:

print f ("Month: May\n");

break;

case 6:

print f ("Month: June\n");

break;

case 7:

print f ("Month: July\n");

break;

case 8:

print f ("Month: August\n");

break;

case 9:

print f ("Month: September\n");

break;

case 10:

print f ("Month: October\n");

break;

case 11:

print f ("Month: November\n");

break;

case 12:

print f ("Month: December\n");

break;

default:

print f ("Invalid month number. Please enter 1 to 12.\n");

}

return 0;

}

**(6). looping in c.**

**->**

**->**

A loop statement allows us to execute a statement or group of statements multiple times based on a condition

-> Types of loops.

1. **. For Loops: -**

It is a repetition control structure that allows you to efficiently write laptop needs to executes a specific number of times.

* Used to efficiently write a loop that needs to execute a specific number of times.
* Syntax:

For (Initialization: Test condition: update);

{

Body of loop;

Update;

}

1. **While Loops: -**

It repeatedly executes a target statement as the long condition is true given.

* It repeatedly executes a target statement as long as the given condition is true.
* Syntax:

While (condition)

{

Body of loop;

Update;

}

**3). Do-While Loops: -**

Do-While loops Is similar to while loop, execute the fact that it Execute once even condition is false.

* Syntax:

Do

{

Body of loop

update;

} while (condition);

**THEORY EXERCISE:**

**->**

**1. while loop**

* **Use when:** You don’t know how many times the loop will run.
* **Example:** Reading input until the user enters a specific value.
* **Code Example**

int n;

scan f ("%d", &n);

while (n! = 0)

{

Print f ("You entered: %d\n", n);

Scan f ("%d", &n);

}

**2. for loop**

* **Use when:** You know how many times the loop should run.
* **Example:** Iterating over an array or counting from 1 to 10.
* **Code Example:**

For (int I = 1; I <= 10; I++)

{

Print f ("%d ", I);

}

**3. do-while loop**

* **Use when:** The loop **must run at least once**, regardless of the condition.
* **Example:** Displaying a menu at least once before user chooses to exit.
* **Code Example:**

int option;

do

{

Print f ("1. Option A\n2. Option B\n0. Exit\n");

Scan f ("%d", &option);

} while (option! = 0);

**LAB EXERCISE:**

->

#include <stdio .h>

int main ()

{

int i;

// Using while loop

Print f ("Using while loop:\n");

I = 1;

while (i <= 10)

{

Print f ("%d ", i);

I++;

}

Print f ("\n");

// Using for loop

Print f ("Using for loop:\n");

for (I = 1; I <= 10; I++)

{

Print f ("%d ", I);

}

Print f ("\n");

// Using do-while loop

Print f ("Using do-while loop:\n");

I = 1;

do

{

Print f ("%d ", i);

I++;

} while (I <= 10);

Print f ("\n");

return 0;

}

**(7). Looping control statements.**

**->**

Looping control statements allow repeated execution of a block of code until a specific condition is met.

**THEORY EXERCISE:**

**->**

1. **Break statement.**

* The break statement is used inside loop or switch statement.
* When compiler finds the break statement inside a loop, compiler will abort the loop and continue to execute followed by loop.
* Syntax: break;

1. **Continue statement.**

* The continue statement is also used inside loop.
* When compiler finds the continue statements inside a loop, compiler will skip all the following statements in the loop resume the next loop iteration.
* Syntax: continue;

1. **Goto statement.**

* By using this Goto statements we can transfer the control from current location to anywhere in the program.
* To do all this have to specify a label with Goto and the control will transfer to the location where the label is specified.
* Syntax:

Goto label;

----------------------

----------------------

Label;

-----------------------

-----------------------

**LAB EXERCISE:**

**->**

#include <stdio .h>

int main ()

{

//use of break statements.

int I;

print f ("Using break (stop at 5): \n");

for (I = 1; I <= 10; I++)

{

If (I == 5)

{

break;

}

Print f ("%d\n", i);

}

// use of continue statements.

Print f ("Using continue (skip 3): \n");

For (I = 1; I <= 5; I++)

{

If (I == 3)

{

continue;

}

Print f ("%d\n", I);

}

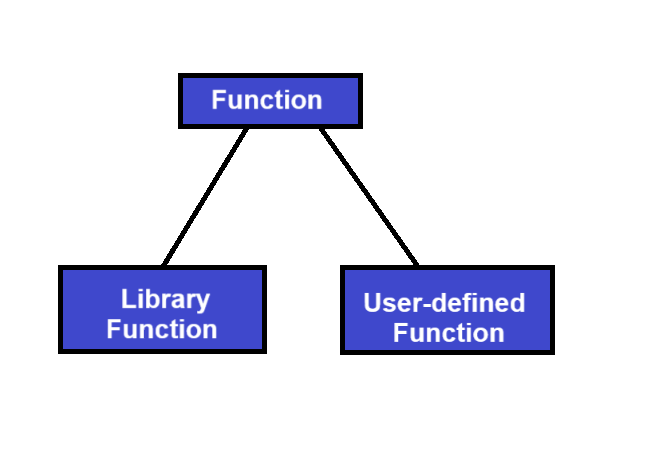
return 0;

}

**(8). Functions in c.**

**->**

A function is a block of organized code that is used to perform a single task. They provide better modularity for your application and reuse-ability. Depending on the programming language, a function may be called a subroutine, a procedure, a routine, a method, or a subprogram.



1. Library Function.

* Special functions inbuilt in c. -> Scan f (), Print f ().

1. User-define Function.

* Declared & defined by programmer.

**THEORY EXERCISE:**

**->**

A function is a block of organized code that is used to perform a single task. They provide better modularity for your application and reuse-ability. Depending on the programming language, a function may be called a subroutine, a procedure, a routine, a method, or a subprogram.

**1.Function Declaration (Prototype): -**  
 Tells the compiler about the function’s name, return type, and parameters (before it is used).

Return \_type function \_name (parameter\_ list);

2. **Function Definition: -**  
 Contains the actual body of the function where tasks are performed.

Return \_type function \_name (parameter\_ list)

{

// statements

return value;

}

3. **Function Call: -**  
 Invokes or executes the function in the main program.

Function \_name (arguments);

Example: -

#include <stdio. h>

// Function declaration

int add (int a, int b);

int main ()

{

int x = 5, y = 3, result;

// Function call

result = add (x, y);

print f ("Sum = %d\n", result);

return 0;

}

// Function definition

int add (int a, int b)

{

return a + b;

}

**LAB EXERCISE:**

**->**

#include <stdio. h>

// Function declaration

long factorial (int n);

int main ()

{

int number;

long result;

// Input from user

Print f ("Enter a positive integer: ");

Scan f ("%d", &number);

// Check for valid input

if (number < 0)

{

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

}

Else

{

// Function call

result = factorial(number);

print f ("Factorial of %d is %d\n", number, result);

}

return 0;

}

// Function definition

long factorial (int n)

{

Long fact = 1;

for (int I = 1; I <= n; I++)

{

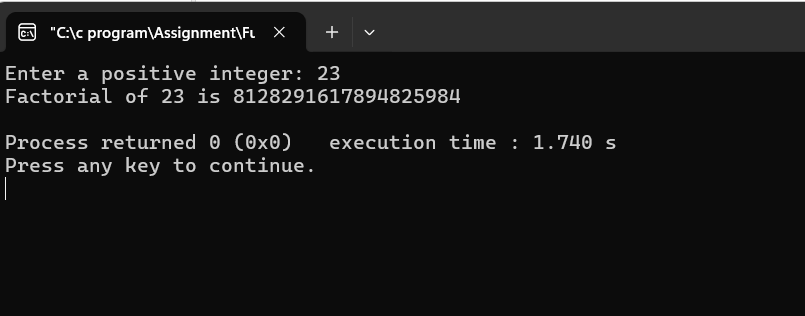
fact \*= I;

}

return fact;

}

Output: -



**(9). Array in c.**

**->** A array is a group of elements. An array is used to collection variables of the same data type, array is ‘0’ based index.

**THEORY EXERCISE:**

**->**

An array in C is a collection of elements of the same data type stored in contiguous memory locations. Arrays allow storing multiple values under a single variable name, making it easier to manage lists of data.

**-> Declaring Arrays.**

* In declaration we specify the type of element and size of the array element.
* **Syntax:**

Data \_type array \_name[size]:

Ex. – int roll [20];

**-> Initializing Arrays**

* We can initialize an array in c either one by one or using single statement
* Ex.

Double balance [--] = {1000.0, 2.0, 3.4, 7.0, 50.0}

OR

Balance [4] = 50.0;

**-> Accessing Array Elements.**

* An element is accessed by placing the index of the element within the square brackets after the name of the array.
* **Ex:** Double income = balance [9];

**-> Types of arrays.**

1. **One-dimensional.**

**->** A one-dimensional array is like a list of values.

**Ex.** Int roll [20];

1. **Multi-dimensional.**

**->** A multi-dimensional array is an array of arrays, commonly used for matrices.

**Ex. int matrix [2] [3] = {{1, 2, 3}, {4, 5, 6}}**

**Print f ("%d", matrix [1] [2]);**

**LAB EXERCISE:**

**->**

#include <stdio. h>

int main ()

// Part 1: One-dimensional array

int arr [5];

print f ("Enter 5 integers:\n");

for (int I = 0; I < 5; I++)

{

Print f ("Element %d: ", i + 1);

Scan f ("%d", & arr [i]);

}

Print f ("\n You entered the following 1D array:\n");

for (int I = 0; I < 5; I++)

{

Print f ("%d ", arr [i]);

}

// Part 2: Two-dimensional array (3x3 matrix)

int matrix [3][3];

int sum = 0;

print f ("\n\n Enter elements for a 3x3 matrix:\n");

for (int I = 0; I < 3; I++)

{

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

{

Print f ("Element [%d] [%d]: ", i, j);

Scan f ("%d", &matrix[i][j]);

sum += matrix[i][j];

}

}

Print f ("\n The 3x3 matrix is:\n");

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

{

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

{

Print f ("%d\t", matrix[i][j]);

}

Print f ("\n");

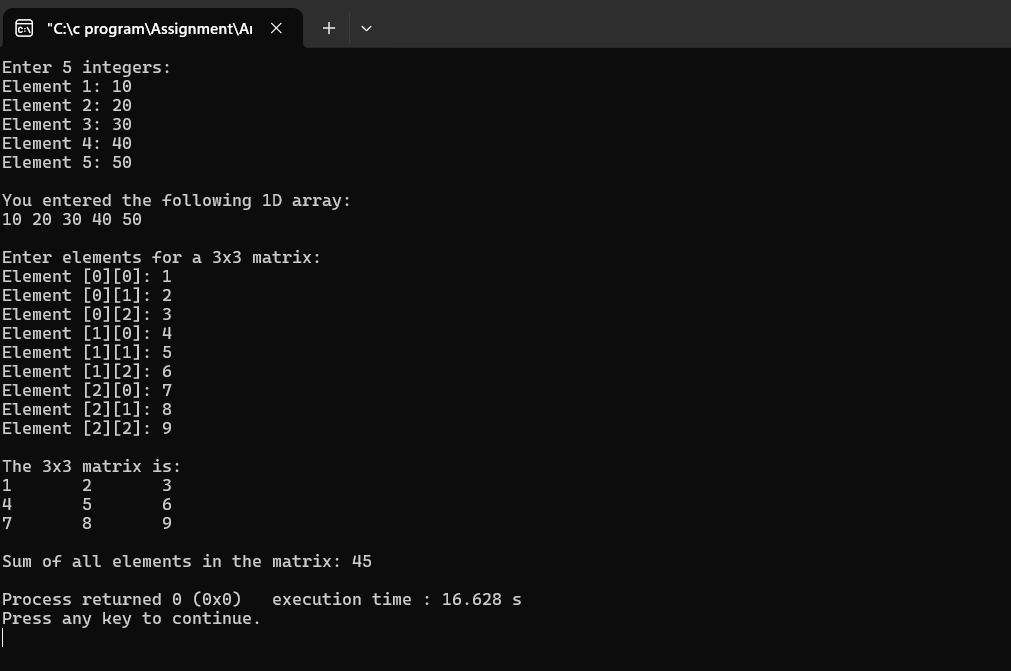
}

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

return 0;

}

**Output: -**

****

**(10). Pointer in c.**

**->**

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

**THEORY EXERCISE:**

**->**

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

**-> Pointer Declaration:**

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

-**> Pointer Initialization:**

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

Int x = 10;

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

access the value of x through \*ptr.

**LAB EXERCISE:**

**->**

#include<stdio.h>

int main ()

{

int num =1520;

int \*p;

p=& num;

print f ("Address of value %x\n", &num);

print f ("Address of pointer %x\n", p);

num=1050;

print f ("After modified value\n");

print f ("Address of pin %x\n", & num);

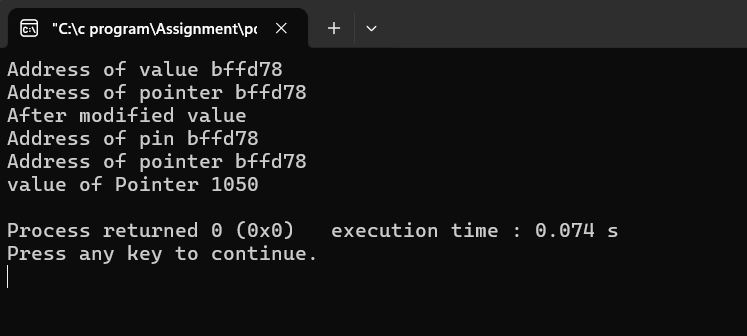
print f ("Address of pointer %x\n", p);

print f ("value of Pointer %d\n", \*p);

return 0;

}

**Output: -**

****

**(11). String in C.**

**->**

* A string is a group of character. String is an array of characters stored in a consecutive memory Locations
* The ending character is always the null character ‘\0’. It acts as string terminator.
* **Syntax:**

Char string \_name [length];

* The compiler automatically places the ‘\0’ at the string when we Initializes the array

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| H | e | l | l | p | \0 |

****



**Null Character**

**THEORY EXERCISE:**

**->**

strings are handled as arrays of characters terminated by a null character ('\0'). The C Standard Library provides several functions to manipulate strings, defined in the header file <string. h>. Here's a detailed explanation of the most common string handling functions:

1. strlen ()

-> Returns the number of characters in a string (excluding the null terminator).

Example:

#include <stdio. h>

#include <string. h>

int main ()

{

char name [] = "Ravi";

print f ("Length = %lu \n", strlen (name));

return 0;

}

1. strlcpy ()

-> Copies one string into another.

Example:

#include <stdio. h>

#include <string. h>

int main ()

{

char source [] = "Ravi";

char destination [20];

strcpy (destination, source);

print f ("Copied String: %s\n", destination);

return 0;

}

1. strcat ()

-> Appends one string to the end of another.

Example:

#include <stdio .h>

#include <string. h>

int main ()

{

char str1[30] = "Ravi ";

char str2[] = "Suthar";

strcat(str1, str2);

print f ("Concatenated String: %s\n", str1);

return 0;

}

1. strcmp ()

-> Compares two strings lexicographically.

Example:

#include <stdio. h>

#include <string .h>

int main ()

{

char a [] = "apple";

char b [] = "banana";

int result = strcmp (a, b);

print f ("Comparison Result: %d\n", result);

return 0;

}

1. strchr ()

-> Searches for the first occurrence of a character in a string.

Example:

#include <stdio. h>

#include <string. h>

int main ()

{

char str [] = "Welcome";

char \*ptr = strchr (str, 'c');

if (ptr! = NULL)

print f ("Character found at position: %ld\n", ptr - str);

else

print f ("Character not found.\n");

return 0;

}

**LAB EXERCISE:**

**->**

#include <stdio. h>

#include <string. h>

int main ()

{

char str1[100],

str2[100];

// Get first string from user

Print f ("Enter first string: ");

Fgets (str1, sizeof (str1), stdin);

// Remove newline character if present

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

// Get second string from user

Print f ("Enter second string: ");

Fgets (str2, sizeof (str2), stdin);

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

// Concatenate strings

Strcat (str1, str2);

// Display result

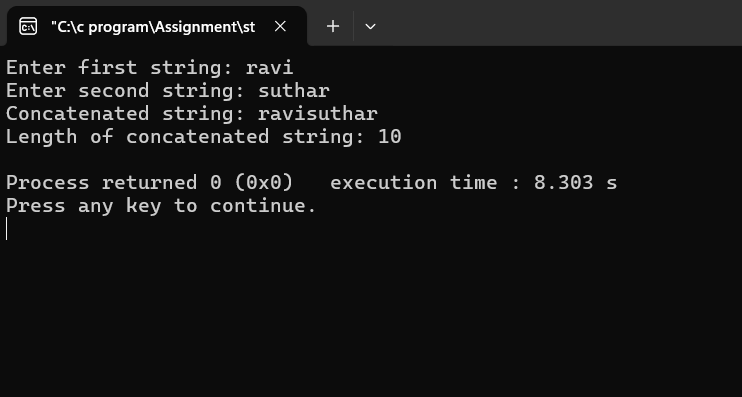
Print f ("Concatenated string: %s\n", str1);

Print f ("Length of concatenated string: %lu \n", strlen (str1));

return 0;

}

**Output: -**

****

**(12). Structure in C.**

**->**

In C, a structure is a user-define data type that allows grouping different types of data under a single name. Each data element in a structure is called a member, and they can be of different types, such as integer, float, or arrays. Structure Are used to represent complex data entities, such as records in databases. They are defined using the struct keyword and can be accessed using the ‘struct’ keyword and can be accessed using the dot operator (‘.’).

**THEORY EXERCISE:**

**->**

In c language, a struct is a user-define data type that groups different data type under a single name. It is declared using the struct keyword, followed by the structure name and member variables. Structure variables can be initialized separ ately (s1.age = 20;), directly (Struct student s2 = {“Alice”, 21, 90.0};), using typedef, or as an array (struct students [2] = {{“Mike”, 19, 78.5}, {“sara”, 20, 92. 0} };). Structures help in organizing complex data efficiently.

**LAB EXERCISE:**

**->**

#include<stdio.h>

#include<string.h>

struct student

{

Char name [20];

int roll;

float cgpa;

};

int main ()

{

//student data 1.

struct student s1;

strcpy (s1.name,"Ravi");

s1. Roll =1229;

s1. cgpa=7.31;

print f (" ===== Student Data =====\n");

print f (" ---- Student 1 ----\n");

print f ("Name is: %s\n", s1 .name);

print f ("Roll no is: %d\n", s1. Roll);

print f ("Cgpa is: %.2f\n", s1. cgpa);

//student data 2.

struct student s2;

strcpy (s2.name,"Kuldip");

s2. Roll =1162;

s2. cgpa =6.63;

print f (" ---- Student 2 ----\n");

print f ("Name is: %s\n", s2.name);

print f ("Roll no is: %d\n", s2. Roll;

print f ("Cgpa is: %.2f\n", s2. cgpa);

//student data 3.

struct student s3;

strcpy (s3.name,"Himmat");

s3. Roll =1192;

s3. Cgpa =7.20;

print f (" ---- Student 3 ----\n");

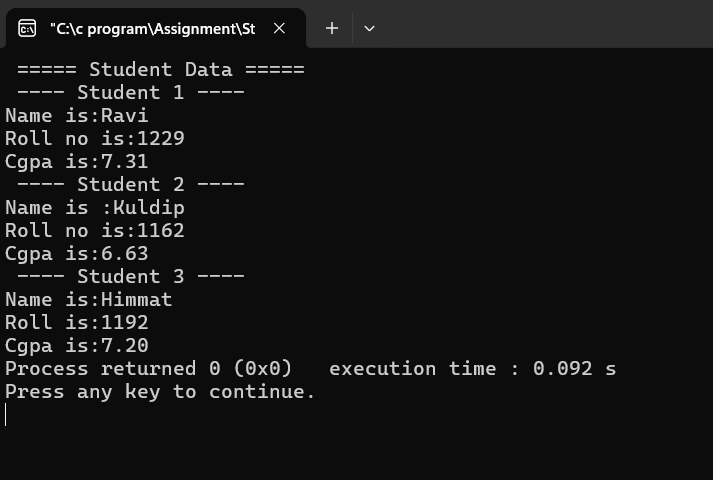
print f ("Name is: %s\n", s3.name);

print f ("Roll is: %d\n", s3. Roll);

print f ("Cgpa is: %.2f", s3. cgpa);

}

**Output: -**

****

**(13). File Handling in C.**

**->**

File handling in C is the process in which we create, open, read, write, and close operations on a file. C language provides different functions such as fopen (), fwrite (), fread (), fseek (), fprintf (), etc. to perform input, output, and many different C file operations in our program.

**THEORY EXERCISE:**

**->**

1. Opening file (fopen())

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

* “r” (read), “w” (write), “a” (append)
* “r+” (read / write), “w+” (write / read), “a+” (append / read)

FILE \*file = fopen (“Data.txt”, “w”); // Opens file in write mode

If (file == NULL)

{

Print f (“Error opening file!\n”);

Return 0;

}

1. Writing to a File (fprintf ())

F print f (file, "Hello, File Handling!"); // Write text to file

F close (file); // Close file after use

1. Reading from a File (F scan f ())

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

char str [100];

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

print f ("Read: %s", str);

f close (file);

1. Closing a File (f close ())

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

**LAB EXERCISE:**

**->**

#include <stdio .h>

int main ()

{

FILE \*fp;

char str [] = "Hello, this is a file I/O example in C!";

char ch;

// Step 1: Create and write to a file

F p = f open ("hello. c", "w"); // Open for writing

if (f p == NULL)

{

Print f ("Error opening file for writing.\n");

return 1;

}

F print f (f p, "%s", str); // Write string to file

F close (f p); // Close the file after writing

Print f ("Data written to file successfully.\n");

// Step 2: Open and read from the file

F p = f open ("hello. c", "r"); // Open for reading

if (f p == NULL)

{

Print f ("Error opening file for reading.\n");

return 1;

}

Print f ("Contents of the file:\n");

while ((c h = f getc (f p))! = EOF)

{

Put char (c h); // Print each character

}

F close (f p); // Close the file after reading

return 0;

}

**Output: -**

