HANNU5

Learn Really Satisfies

PROGRAMMING FOR PROBLEM SOLVING

By : KONGARI UMA

Kongariuma69@gmail.com

Programming for Problem Solving

Important Long Questions

UNIT - I:

1. Explain the structure of a C program.
2. Explain about operators in C.
3. Explain about tokens in C.
4. Explain about data types in C.
5. Describe the various selection statements available in C.
6. Describe the various loop statements available in C.
7. Explain formatted I/O functions in C.
8. Explain Decision Steps in Algorithms

UNIT - II:

1. What is top-down design in programming? Explain with an example.
2. What are library functions? Give three examples used in C.
3. Explain functions with input arguments and functions without arguments.
4. Explain pointers and the indirection (*) operator with an example.
5. How do you pass input&output parameters to a function using pointers?
6. What is the scope of a variable in C? Differentiate between local and global variables.
7. Explain formal output parameters as actual arguments with an example.
8. Write a program using a function to swap two numbers using pointers.
9. Explain modular programming and its advantages.

UNIT – III

1. Explain array subscripts and sequential access using a for loop.
2. How do you declare and initialize an array in C?
3. How can array elements be passed as arguments to functions?
4. Write a program to search for an element in an array.
5. Explain sorting. Discuss different sorting techniques in detail.?
6. What are parallel arrays? Give an example.
7. Explain multidimensional arrays with an example of a 2D array.
8. How are strings stored in C? Explain null-termination.
9. Explain string library functions?
10. How can you create an array of pointers to strings in C?

UNIT - IV

1. What is recursion? Explain recursion with an example of a factorial function.
2. How do you trace a recursive function? Explain with a flow diagram.
3. Write a recursive function to calculate the Fibonacci series.
4. Explain the use of arrays and strings in recursive functions.
5. What are structures in C? Give an example.
6. How can structure data be passed as input/output parameters to?
7. Explain functions with structured result values using struct.
8. Give an example of a union and explain how memory is shared.

UNIT - V

1. Explain the difference between text and binary files in C.
2. How do you open, read, and write a file in C?
3. What are file pointers? Explain fopen, fclose, fread, and fwrite.
4. Write a program to read data from a text file and display it.
5. Explain linear search and implement it for an array of integers.
6. Explain binary search and implement it for a sorted array.
7. Explain bubble sort and write a program to sort an array.
8. Explain insertion sort and give a simple program.
9. Explain selection sort with an example program.
10. What are the time complexities of insertion sort, bubble sort, and selection sort?

Important Short Questions and Answers

UNIT I

C Language Basics, Control Structures, and Loops

1. What are the basic elements of C language?

The basic elements of C are keywords, identifiers, constants, variables, operators, expressions, and statements, which together are used to write and execute C programs.

2. How do you declare variables in C?

A variable declaration specifies the data type and name of a variable

for example, int age;

and it can also be initialized during declaration

such as int x = 10;.

3. What are the basic data types in C?

The basic data types in C are int (integers), float (decimal numbers), double (double-precision decimals), char (single character), and void (no value).

4. What is an executable statement?

An executable statement is a line of code that performs an action during program execution,

such as printf("Hello World"); or x = y + 5;.

5. Give the general form of a C program.

A C program consists of preprocessor directives, the main() function, declarations, executable statements, and a return statement.

Example

```
#include <stdio.h>
int main() {
    int a, b;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    printf("Sum = %d", a + b);
    return 0;
}
```

6. What is an arithmetic expression?

An arithmetic expression is a combination of **constants, variables, and operators** that computes a value.
Example:

```
int a = 10, b = 5;
int sum = a + b;   // addition
int diff = a - b;  // subtraction
```

7. How can you format numbers in output?

printf() uses **format specifiers** to control how numbers are displayed:

- %d → integer
- %f → floating-point number
- %.2f → float with 2 decimal places

Example:

```
float avg = 12.3456;
printf("Average = %.2f", avg); // Output: Average = 12.35
```

8. What is a selection structure?

A selection structure allows the program to **choose between alternatives** based on a condition.
Example:

```
if(score >= 50)
    printf("Pass");
else
    printf("Fail");
```

The program executes different statements depending on the condition.

9. Difference between if and if-else.

1. if executes a block of code **only when the condition is true**.
2. if-else provides an **alternative block** if the condition is false.

Example:

```
if(a > b)
   printf("A is greater");
else
   printf("B is greater or equal");
```

10. Difference between for, while, and do-while loops.

Loop Type	When to Use	Key Feature
for	Known number of iterations	Initialization, condition, and increment/decrement in one line
while	Unknown number of iterations	Condition checked before executing loop body
do-while	Executes at least once	Condition checked after executing loop body

UNIT II

Functions and Pointers

1. What is a function in C?

A function is a **self-contained block of code** designed to perform a specific task. Functions make programs modular, easy to read, and reusable. Every C program has at least one function: main().

2. What are the types of functions in C?

Functions are categorized based on arguments and return values:

1. Without arguments and without return value
2. With arguments and without return value
3. With arguments and with return value
4. Without arguments and with return value

3. What is a library function?

A library function is a **predefined function** provided by C, which saves time because we do not need to write code for common tasks.
Examples:

printf() → prints output

scanf() → reads input

4. What is top-down design in programming?

Top-down design is a method of **breaking a large program into smaller, manageable modules or functions**. Each module handles a specific sub-task. This approach improves readability, debugging, and maintenance.
Example: For a payroll program:

- Module 1: Read employee details
- Module 2: Calculate salary
- Module 3: Print payslip

5. What is a pointer in C?

A pointer is a **variable that stores the address of another variable**. It allows direct access to memory locations, which is useful for dynamic memory allocation, arrays, and function arguments.

6. What is the indirection operator * used for?

The * operator, also called the **dereference operator**, is used to access the **value stored at the memory address** a pointer holds.

7. Give an example of a function with input arguments.

A function can accept values (arguments) from the calling function.
Example:

```
void display(int n) {
   printf("The number is %d", n);
}
int main() {
   display(5); // Pass 5 as argument
   return 0;
}
```

Here, 5 is passed to display() as input.

8. What is the scope of a variable?

Scope defines **where a variable can be accessed** in a program.

1. **Local scope:** Variable declared inside a function; accessible only within that function.

```
void f() { int x = 5; } // x is local to f()
```

2. **Global scope:** Variable declared outside all functions; accessible throughout the program.

```
int x = 10; // Global variable
void f() { printf("%d", x); }
```

9. What is modular programming?

Modular programming involves **dividing a program into independent functions or modules**, each performing a specific task.

Example:

```
void readData() { /* read input */ }
void calculate() { /* compute results */ }
void display() { /* print output */ }
```

10. How do you pass arrays to functions in C?

Arrays are passed to functions **by reference**, meaning the function can access and modify the original array elements.
Example:

```
void printArray(int arr[], int n) {
   for(int i=0; i<n; i++)
      printf("%d ", arr[i]);
}
int main() {
   int nums[] = {1, 2, 3, 4};
   printArray(nums, 4);
   return 0;
}
```

Here, printArray prints elements of the array nums. Any modification in the function affects the original array.

UNIT III

Arrays and Strings

1. How do you declare an array in C?

An array is a **collection of elements of the same data type**, stored in contiguous memory locations. To declare an array, specify the data type, name, and size:

```
datatype array_name[size];
```

Example:

```
int marks[5];  // Array of 5 integers
float prices[10]; // Array of 10 floats
```

Here, marks can store 5 integer values: marks[0] to marks[4].

2. How do you access array elements?

Each element of an array is accessed using an **index (subscript)**, starting from 0.
Example:

```
marks[0] = 90;
marks[1] = 85;
printf("%d", marks[0]); // Prints 90
```

marks[0] is the first element, marks[1] is the second, and so on.

3. How do you pass an array to a function?

Arrays are passed **by reference**, so the function can access or modify the original array.
Example:

```
void printArray(int arr[], int n) {
   for(int i=0; i<n; i++)
       printf("%d ", arr[i]);
}

int main() {
   int nums[] = {1, 2, 3, 4};
   printArray(nums, 4);
   return 0;
}
```

Here, arr[] in the function refers to the same memory as nums in main().

4. What is a multidimensional array?

A multidimensional array has **more than one dimension** and is used to store data in **rows and columns**.
Example of a 2D array:

```
int matrix[2][3] = { {1,2,3}, {4,5,6} };
printf("%d", matrix[1][2]); // Prints 6
```

matrix[1][2] refers to the element in **2nd row, 3rd column**.

5. What is a string in C?

A string is an **array of characters ending with a null character \0**.
Example:

```
char name[10] = "Alice";
printf("%s", name); // Prints Alice
```

The null character \0 indicates the end of the string.

6. What are some common string library functions?

The C library provides functions to manipulate strings:

1. strcpy(dest, src) → copies src to dest
2. strcat(dest, src) → concatenates src at the end of dest
3. strlen(str) → returns the length of the string
4. strcmp(str1, str2) → compares two strings, returns 0 if equal

7. How do you concatenate strings in C?

Concatenation joins two strings together using the strcat() function:

```
char s1[20] = "Hello, ";
char s2[] = "World!";
strcat(s1, s2);
printf("%s", s1); // Prints "Hello, World!"
```

8. What is an array of pointers?

An array of pointers stores **addresses of strings**. Useful for storing multiple strings of variable length.

9. What is the difference between arrays and pointers in C?

An array stores multiple elements of the same type in contiguous memory locations, while a pointer stores the address of a variable or memory location.

10. How do you find the length of a string?

Use the strlen() function from string.h, which counts the number of characters **excluding the null character**.
Example:

```
char str[] = "Hello";
int len = strlen(str);
printf("Length = %d", len); // Prints 5
```

UNIT IV

Recursion, Structure, and Union

1. What is recursion in C?

Recursion occurs when a **function calls itself** to solve a smaller instance of a problem. It is often used in mathematical problems, data structures, and algorithms.

2. What are the advantages of recursion?

1. Simplifies complex problems (e.g., factorial, Fibonacci, tree traversal)
2. Reduces code length
3. Easy to understand and debug for mathematical algorithms

Disadvantage: May consume more memory due to multiple function calls (stack usage).

3. Give an example of a recursive mathematical function.

Fibonacci series using recursion:

```
int fibonacci(int n) {
   if(n == 0) return 0;
   if(n == 1) return 1;
   return fibonacci(n-1) + fibonacci(n-2);
}
```

fibonacci(5) returns 5 (series: 0,1,1,2,3,5…)

4. What is a structure in C?

A structure is a **user-defined data type** that groups variables of different types under a single name.

5. How do you pass a structure to a function?

Structures can be passed **by value** (copy) or **by reference** (using pointers).
Example by value:

```
void display(struct Student s) {
   printf("%s %d %f", s.name, s.roll, s.marks);
}
```

Example by reference (pointer):

```
void display(struct Student *s) {
   printf("%s %d %f", s->name, s->roll, s->marks);
}
```

6. What are union types in C?

A union is similar to a structure, but **all members share the same memory location**, so only **one member can store a value at a time**.

Types of Unions in C:

- Simple Union
- Nested Union
- Anonymous Union

7.Difference between structure and union

Feature	Structure	Union
Memory	Each member has separate memory	All members share same memory
Storage	Can store all members simultaneously	Only one member can hold value at a time
Syntax	struct Name { ... }	union Name { ... }
Use	When multiple data needed	When only one data type needed at a time

8. How can functions return structure values?

Functions can return structures just like normal data types.
Example:

```
struct Point {
    int x, y;
};
struct Point getOrigin() {
    struct Point p = {0, 0};
    return p;
}
```

getOrigin() returns a structure with x=0 and y=0.

9. What are recursive functions with array and string parameters?

Recursive functions can also **process arrays and strings**.
Example: Sum of elements in an array:

```
int sumArray(int arr[], int n) {
    if(n == 0) return 0;
    return arr[n-1] + sumArray(arr, n-1);
}
```

sumArray([1,2,3,4],4) → 10

10. Why use structures and unions in C?

1. **Structures**: Organize related data of different types; pass complex data to functions easily
2. **Unions**: Save memory when only **one value is needed** at a time
3. Improve code readability and maintainability
4. Useful in real-world applications like databases, records, and memory-efficient programs.

UNIT V

Files, Searching, and Sorting

1. What is a text file in C?

A text file stores data as **human-readable characters** (ASCII). You can read or write data using standard file operations.

2. What is a binary file in C?

A binary file stores data in **machine-readable binary format**, not directly readable by humans. It is faster for reading/writing large amounts of data.

3. Name common file I/O functions in C.

1. fopen() → open a file
2. fclose() → close a file
3. fprintf() → write formatted text to a file
4. fscanf() → read formatted data from a file
5. fread() → read binary data
6. fwrite() → write binary data

4. What is linear search?

Linear search checks each element sequentially until the target is found. Works on unsorted arrays.

5. What is binary search?

Search finds a target **in a sorted array** by repeatedly dividing the search interval in half.

6. What is bubble sort?

Bubble sort **repeatedly swaps adjacent elements** if they are in the wrong order.

7. What is insertion sort?

Insertion sort builds a **sorted array one element at a time** by inserting the next element into the correct position.

8. What is selection sort?

Selection sort repeatedly **selects the smallest element** from unsorted part and places it at the beginning.

9. Difference between linear and binary search

Feature	Linear Search	Binary Search
Array must be sorted	No	Yes
Time complexity	O(n)	O(log n)
Method	Sequentially checks each element	Divides array in half repeatedly
Use	Small or unsorted data	Large sorted data

10. Why use binary files?

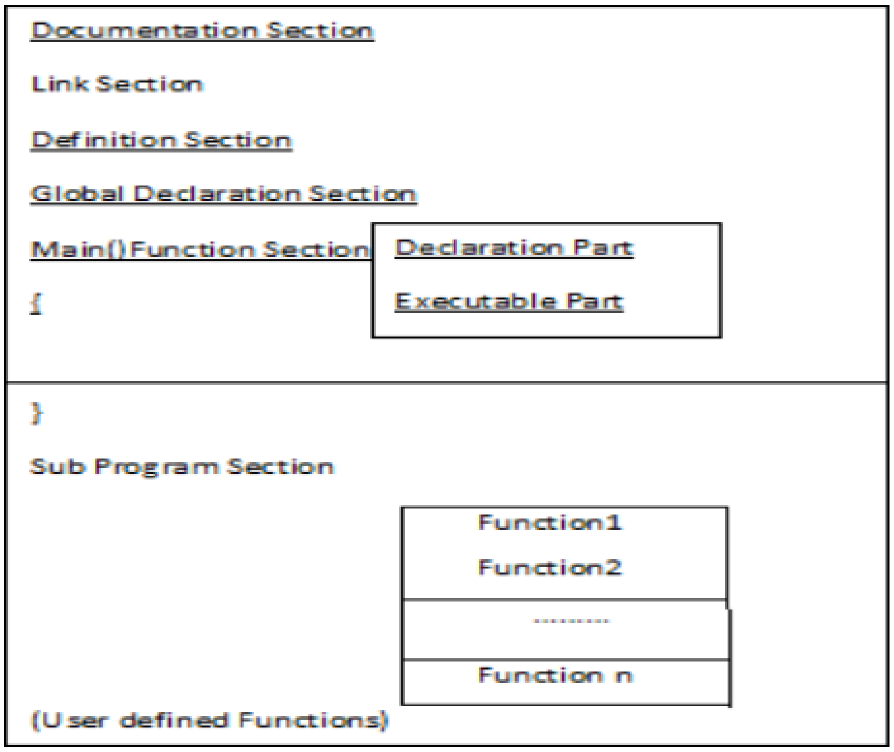
1. Faster read/write for large data
2. Saves memory by storing in compact form
3. Used in databases, images, and executable files
4. Supports storing complex structures and arrays efficiently

Important Long Questions and Answers

UNIT - I

1. Explain the structure of a C program.

Ans: Structure of C program:



i) Documentation Section: This section contains **comments** that explain the purpose and functionality of the program. Comments improve the readability of the code and are ignored by the compiler during execution.
Ex:- /*Addition of two numbers */
// Addition of two numbers

(ii) Link (Pre-processor / Header file) Section: This section includes header files using the #include directive, which brings in standard libraries or other resources needed by the program. It is placed at the beginning of the program.

Example:

```
#include <stdio.h>  // Standard Input Output library
#include <stdlib.h> // Standard library
```

(iii) Definition Section: This section defines **macros** using #define. Macros are constants or code snippets replaced by their values before compilation, often used for constant values.

Example:

```
#define PI 3.14159
#define MAX 100
```

(iv) Global Declaration Section

Definition:
This section contains declarations of global variables and function prototypes that are accessible throughout the program, including inside all functions.

Example:

```
int counter;              // Global variable
int increment(int value);  // Function prototype
```

(v) Main Program Section: This section contains the main() function, which is the entry point of the program where execution starts. It usually controls the flow of the program.

Example:

```
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

(v) Sub-program Section: This section contains user-defined functions other than main(). These functions perform specific tasks and help break the program into smaller, manageable pieces.

Example:

```
int calculate_average(int x, int y, int z) {
    return (x + y + z) / 3;
}
```

Example program to add two numbers and display the result

1. Documentation Section
   Program to add two numbers and display the result
   */

```
#include <stdio.h>  // 2. Link Section

// 4. Global Declaration Section
int add(int a, int b);  // Function prototype

// 5. Main Program Section
int main() {
    int num1, num2, sum;

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

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

    sum = add(num1, num2);

    printf("Sum = %d\n", sum);

    return 0;
}

// 6. Sub-program Section
int add(int a, int b) {
    return a + b;
}
```

2. Explain about operators in C.

C operators can also be classified based on their function:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment and Decrement Operators
6. Conditional (Ternary) Operator
7. Bitwise Operators
8. Special Operators

1. **Arithmetic Operators:** C provides all the basic arithmetic operators. These can operate on any built –in data type allowed in C.

The arithmetic operators are listed as follows:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo division

2. Relational Operators:

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0. The following table shows all relation operators supported by C.

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than t
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Ex:- 4.5<=10(true)

6.5<-10(false) 10<4+12(true)

3.Logical Operators

Logical operators are used to test **multiple conditions**.

(a) Logical AND (&&)

Returns true only if **both conditions are true**.

Condition 1	Condition 2	Result
true	True	True
true	False	False
false	True	False
false	False	False

(b) Logical OR (||)

Returns true if **any one condition is true**.

Condition 1	Condition 2	Result
true	True	True
true	False	True
false	True	True
false	False	False

(c) Logical NOT (!)

Reverses the logical value of an expression.

Condition	Result
true	False
false	True

4.Assignment Operators

Assignment operators assign a value, expression, or variable to another variable.

Syntax:

variable = value/expression/variable;

Examples:

```
x = 10;
y = a + b;
z = p;
```

Compound Assignment Operators

Operator	Example	Meaning
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

5.Increment and Decrement Operators

These are unary operators used to increase or decrease a variable's value by **1**.

Increment Operator (++)

- **Pre-increment (++x)**: Increments before use

```
int x = 5;
printf("%d", ++x);  // Output: 6
```

- **Post-increment (x++)**: Uses value first, then increments

```
int x = 5;
printf("%d", x++);  // Output: 5
// x becomes 6
```

Decrement Operator (--)

- **Pre-decrement (--x)**: Decrements before use

```
printf("%d", --x);  // Output: 4
```

- **Post-decrement (x--)**: Uses value first, then decrements

```
printf("%d", x--);  // Output: 5
// x becomes 4
```

6. Conditional (Ternary) Operator

The **conditional operator**, also known as the **ternary operator**, operates on **three operands**. It evaluates a given condition and returns **one of two values** based on whether the condition is true or false.

Syntax:

```
(condition) ? expression_if_true : expression_if_false;
```

7. Bitwise Operators

Bitwise operators perform operations at the **bit level**.

Operator	Name	Description
&	Bitwise AND	1 if both bits are 1
`	`	Bitwise OR
^	Bitwise XOR	1 if bits are different
~	Bitwise NOT	Inverts bits
<<	Left Shift	Shifts bits left
>>	Right Shift	Shifts bits right

8. Special operators

C supports some special operators such as comma operator, size of operator, pointer operators (& and *) and member selection operators (. and ->).
comma operator: The comma operator is used to link the related expressions together.
sizeof operator: The sizeof is a compile time operator and when used with an operand, it returns the number of bytes the operand occupies. The operand may be variable, a constant or a data type qualifier.

PROGRAM TO DEMONSTRATE C OPERATORS

```
#include <stdio.h>

int main() {
    int a = 10, b = 3;
    int result;

    printf("Arithmetic Operators:\n");
    printf("a + b = %d\n", a + b);
    printf("a - b = %d\n", a - b);
    printf("a * b = %d\n", a * b);
    printf("a / b = %d\n", a / b);
    printf("a %% b = %d\n\n", a % b);

    printf("Relational Operators:\n");
    printf("a > b = %d\n", a > b);
    printf("a < b = %d\n", a < b);
    printf("a == b = %d\n\n", a == b);

    printf("Logical Operators:\n");
    printf("(a > b && a != 0) = %d\n", (a > b && a != 0));
    printf("(a < b || b > 0) = %d\n", (a < b || b > 0));
    printf("!(a == b) = %d\n\n", !(a == b));

    result = a;
    result += b;
    printf("Assignment Operator (result += b): %d\n\n", result);

    printf("Unary Operators:\n");
    printf("++a = %d\n", ++a);
    printf("--b = %d\n\n", --b);

    printf("Bitwise Operators:\n");
    printf("a & b = %d\n", a & b);
    printf("a | b = %d\n", a | b);
    printf("a ^ b = %d\n", a ^ b);
    printf("~a = %d\n\n", ~a);

    int max = (a > b) ? a : b;
    printf("Ternary Operator (max): %d\n\n", max);

    printf("Sizeof Operator:\n");
    printf("Size of int = %lu bytes\n", sizeof(int));

    return 0;
}
```

OUTPUT

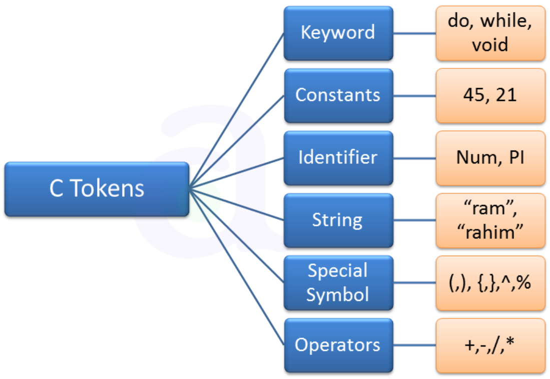
```
Arithmetic Operators:
a + b = 13
a - b = 7
a * b = 30
a / b = 3
a % b = 1

Relational Operators:
a > b = 1
a < b = 0
a == b = 0

Logical Operators:
(a > b && a != 0) = 1
(a < b || b > 0) = 1
!(a == b) = 1

Assignment Operator (result += b): 13

Unary Operators:
++a = 11
--b = 2

Bitwise Operators:
a & b = 2
a | b = 11
a ^ b = 9
~a = -12

Ternary Operator (max): 11

Sizeof Operator:
Size of int = 4 bytes
```

3.Explain about tokens in C.

A token in C can be defined as the smallest individual element of the C programming language that is meaningful to the compiler. It is the basic component of a C program.

Types of Tokens in C:

The tokens of C language can be classified into six types based on the functions they are used to perform. The types of C tokens are as follows:



1. **Keywords**
2. **Identifiers**
3. **Constants**
4. **Strings**
5. **Special Symbols**
6. **Operators**

1.Keywords

A keyword is a reserved word. All keywords have fixed meaning that means we cannot change. Keywords serve as basic building blocks for program statements. All keywords must be written in lowercase.

A list of 32 keywords in c language is given below:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

2.Identifiers

Identifiers refer to the names of variables, constants, functions and arrays. These are user-defined names is called Identifiers. These identifier are defined against a set of rules.

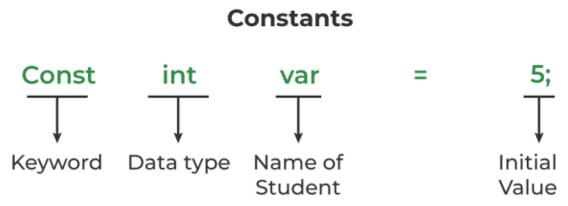
Rules for an Identifier

- An Identifier can only have alphanumeric characters(a-z , A-Z , 0-9) and underscore(_).
- The first character of an identifier can only contain alphabet(a-z , A-Z) or underscore (_).
- Identifiers are also case sensitive in C. For example name and Name are two different identifier in C.
- Keywords are not allowed to be used as Identifiers.
- **Special characters** such as ;, ., whitespace, /, ,, etc., are **not allowed**.
- Identifier may be of reasonable length of 8-10 characters though certain computers allow up to 31 characters

Valid Identifiers	Invalid Identifiers	Reason
Name	123name	Starts with a digit
Name	name!	Contains special character !
_count	user-name	Contains special character -
total123	int	Keyword in C
user_name	my var	Contains whitespace
A1B2C3	#value	Contains special character #
data_2025		

3.Constants

These are fixed values that cannot be changed during the execution of program. Constants can be of different types, such as integer constants, floating-point constants, character constants, and string constants.



Constant	Example
Decimal Constant	10, 20, 450 etc.
Real or Floating-point Constant	10.3, 20.2, 450.6 etc.
Octal Constant	021, 033, 046 etc.
Hexadecimal Constant	0x2a, 0x7b, 0xaa etc.
Character Constant	'a', 'b', 'x' etc.
String Constant	"c", "c program", "c in javatpoint" etc.

4.Strings

These are sequences of characters enclosed in double quotes. Strings can b used to store and manipulate text in a C program.

Example:

char greeting[] = "Hello, World!";

Here, greeting is a string variable holding the text "Hello, World!".

5.Special Symbols

Special symbols in C are characters with predefined meanings that are used to structure programs, separate statements, and perform specific operations. Examples include ;, { }, ( ), [ ], #, ., and ->.

6.Operators

An **operator** is a symbol that performs a specific operation on one or more operands (variables, constants, or expressions) and produces a result.

Classification of Operators in C

Operators in C can be classified **based on the number of operands** and **their functionality**.

Based on Number of Operands

(a) Unary Operators

Unary operators operate on **a single operand**.

Examples:

- Unary plus (+)
- Unary minus (-)
- Increment (++)
- Decrement (--)
- Logical NOT (!)
- Bitwise NOT (~)
- Address operator (&)

(b) Binary Operators

Binary operators operate on **two operands**.

Examples:
Arithmetic, Relational, Logical, Assignment, and Bitwise operators.

(c) Ternary Operator

The **ternary (conditional) operator** operates on **three operands**.

Syntax:

(condition) ? expression1 : expression2;

(d) Special Operators

Special operators perform specific tasks such as memory handling and structure access.

4.Explain about data types in C.

A data type **is a classification that specifies the** kind of value a variable can store**, the** operations that can be performed **on it, and the** amount of memory **allocated. It tells the compiler how to** interpret and store data **in memory, such as integers, floating-point numbers, or characters.**

C data types are mainly classified into **three categories**:

1. Fundamental (Primary) Data Types
2. Secondary (Derived) Data Types
3. User-Defined Data Types

1. Fundamental / Primary Data Types

These are the **basic data types** provided by the C language.

(a) Integer Types

Used to store **whole numbers**.

Examples:

int x = 10;

```
short y = -20;
long z = 1234567890L;
```

- Size of int depends on the processor (2, 4, or 8 bytes).
- Range (2 bytes): –32,768 to +32,767
- Range (4 bytes): –2,147,483,648 to +2,147,483,647

(b) Floating-Point Types

Used to store **decimal numbers**.

Examples:

```
float a = 3.14;
double b = -0.123456;
```

(c) Character Type

Used to store a **single character**.

Example:

```
char c = 'A';
```

(d) Void Type

Represents the **absence of a value**. Commonly used for functions that do not return any value.

Size and Range of Data Types

Data Type	Size (bytes)	Range
char	1	–128 to 127
unsigned char	1	0 to 255
int	2	–32768 to 32767
unsigned int	2	0 to 65535
long int	4	–2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308

2. Secondary / Derived Data Types

These data types are **derived from primary data types**.

(a) Arrays

An **array** is a collection of elements of the **same data type** stored in **contiguous memory locations**.

Example:

```
int a[] = {1, 2, 3, 4, 5};
char str[20] = "hello";
```

Array indexing starts from **0**.

(b) Pointers

A **pointer** is a variable that stores the **address of another variable**.

Example:

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

3. User-Defined Data Tyes

These data types allow users to create **custom data types**.

(a) Structure (struct)

Used to group **variables of different data types**.

Example:

```
struct student {
    char name[50];
    int age;
    float cgpa;
};
```

(b) Union (union)

Similar to structures, but **all members share the same memory location**.

Example:

```
union data {
    int x;
    float y;
    char z[4];
};
```

(c) Enumeration (enum)

Used to define a set of **named integer constants**.

Example:

```
enum colour {black, blue, green, red, yellow};
```

Here, the constants have values **0, 1, 2, 3, and 4** respectively.

5.Describe the various selection statements available in C.

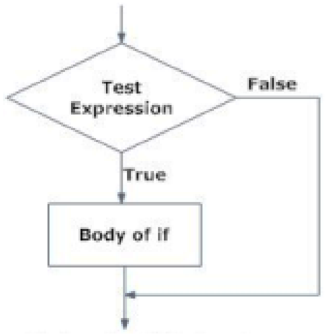
Conditional statements in C are used to make **decisions based on conditions**. They allow the program to **alter the flow of execution** depending on whether a condition is true or false, enabling **decision-making** in a program..

if Statement: An if statement in C checks a condition; if it's true, the code inside runs, otherwise it's skipped.

The general syntax

```
if (condition) {
    // statements
}
```

Flowchart of if statement:



program to check if a person is eligible to vote or not

```
#include <stdio.h>

int main() {
    int num = 5;

    if (num > 0) {
        printf("Number is positive");
    }

    return 0;
}
```

Output:

```
Number is positive
```

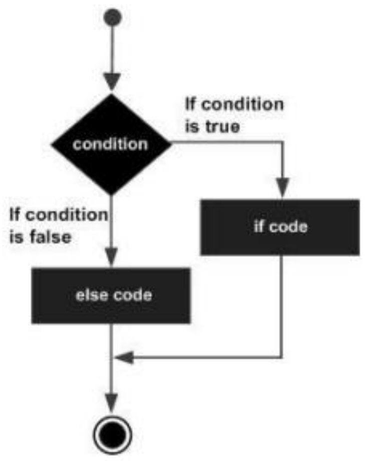
If else statement

An **if-else** statement in C executes one block if the condition is true, otherwise it executes the **else** block.

syntax

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

Flowchart:



Program:

C program to find whether the given number is even or odd*/

```
#include<stdio.h>
#include<conio.h>
void main()
intnum;
clrscr();
printf("Enter a number: ");
scanf("%d", &num);
if (num % 2 == 0)
printf("\n The given number is even");
else
```

```
printf("\nThe given number is odd");
getch();
}
```

Output:

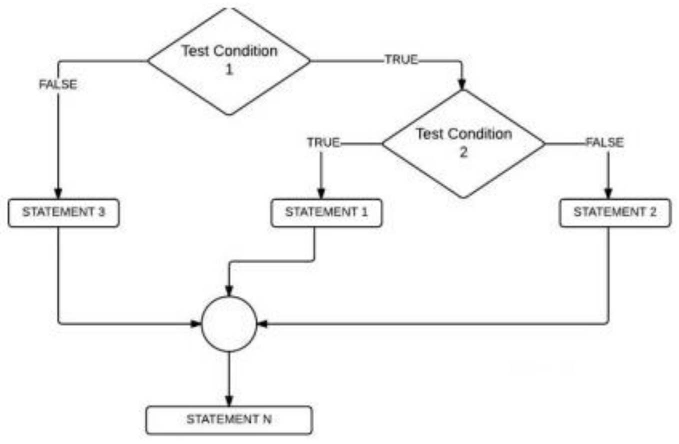
Case: 1

```
Enter a number: 10
The given number is even
```

Nested if....else statement: A nested if-else statement places one if-else inside another to handle multiple conditions, like multi-level security checks at an airport.

Syntax:

```
if (condition1) {
    if (condition2) {
        // statements if both conditions are true
    } else {
        // statements if condition1 is true and condition2 is false
    }
} else {
    // statements if condition1 is false
}
```



Program:

```
#include <stdio.h>

int main() {
    int a = 10, b = 5;

    if (a > b) {
        if (b > 0) {
            printf("a is greater than b and b is positive");
        }
    }
    return 0;
}
```

Output:

```
a is greater than b and b is positive
```

Else-If Ladder

in C is a control flow structure used when you need to test **multiple conditions** one after another. It helps in selecting **one block of code** among many options to be executed.

Syntax of else-if Ladder in C

```
if (condition1) {
    // Block 1: executes if condition1 is true
}
else if (condition2) {
    // Block 2: executes if condition2 is true
}
else if (condition3) {
    // Block 3: executes if condition3 is true
}
...
else {
    // Default block: executes if none of the above conditions are true
}
```

Example Program

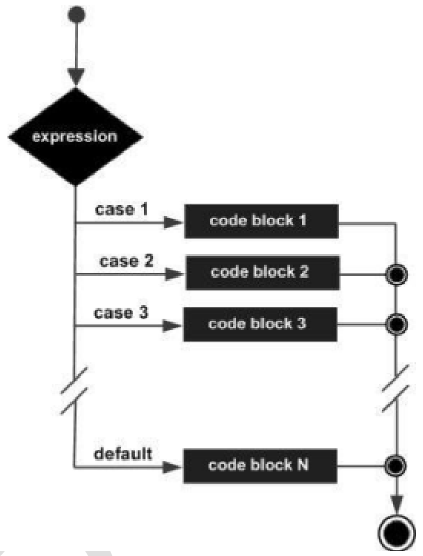
```
#include <stdio.h>

int main() {
    int num;

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

    if (num > 0) {
        printf("The number is positive.\n");
    }
    else if (num < 0) {
        printf("The number is negative.\n");
    }
    else {
        printf("The number is zero.\n");
    }
    return 0;
}
```

Switch statement:

The switch statement in C compares a variable with multiple cases and executes the matching case; if no match is found, the default case runs.

Syntax:

```
        switch( expression )
        {
               case value-1:
                              Block-1;
                              Break;
               case value-2:
                              Block-2;
                              Break;
               case value-n:
                              Block-n;
                              Break;
               default:
                              Block-1;
                              Break;
        }
        Statement-x;
```



C program to print day of week name using switch...case*/

```
#include <stdio.h>
int main() {
int week;
printf("Enter week number (1-7): ");
scanf("%d", &week);
switch(week)
{
case 1:
printf("Its Monday.\n");
break;
case 2:
printf("Its Tuesday.");
break;
case 3:
printf("Its Wednesday.");
break;
case 4:
printf("Its Thursday.\n");
break;
case 5:
printf("Its Friday.\n");
break;
Case
```

printf("Its Saturday. \n")
case 7:
printf("Its Sunday. \n"):
break:
default: printf("Please enter week number between 1-7.");
return 0;
}

Output:

```
Enter week number (1-7):
            6    Its Saturday.
```

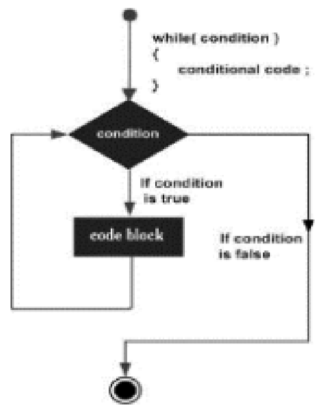
6.Describe the various loop statements available in C.

Loops in C repeat a block of code multiple times until a specified condition becomes false, making repetitive tasks easier.

1.While loop: The while is an entry-controlled loop statement. The while statement is used when the program needs to perform repetitive tasks.

Syntax of While Loop in C:

```
While(condition)
{
Statements;
}
```



- If a condition is true then and only then the body of a loop is executed.
- After the body of a loop is executed then control again goes back at the beginning, and the condition is checked if it is true, the same process is executed until the condition becomes false.
- Once the condition becomes false, the control goes out of the loop

Program:

```
#include<stdio.h>
 #include<conio.h>
 int main()
{
 int num=1; //initializing the variable
 while(num<=10) //while loop with condition
{ printf("%d\t",num);
 num++; //incrementing operation
}
 return 0;
 }
```
Output: 1 2 3 4 5 6 7 8 9 10

2. Do...while loop: The do-while loop in C is a type of exit-controlled loop that repeated executes a block of code, and then checks a condition after each iteration. The de while loop is used when we want to execute the loop body at least once.

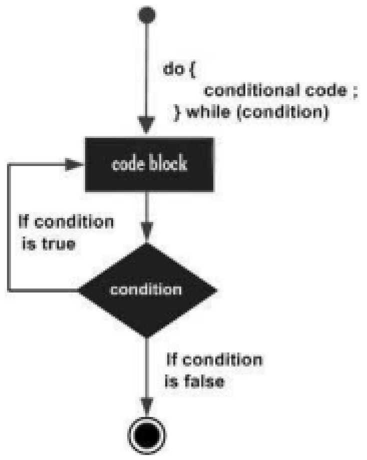
Points to Remember:

1. The loop body is executed at least once.
2. The condition is checked after executing the loop body once.
3. If the condition is false then the loop is terminated.

Syntax:

```
do
{
 statements
}
while (expression);
```

Flow chart:



Program to print the numbers from 1 to 10 using do...while loop"

```
#include<stdio.h>
void main()
{
int i=1;
do
{
printf("%d\t",i);
i=i+1;
}while(i<=10);
printf("\n");
}
```

Output:

```
            1     2   3   4   5  6  7  8  9  10
```

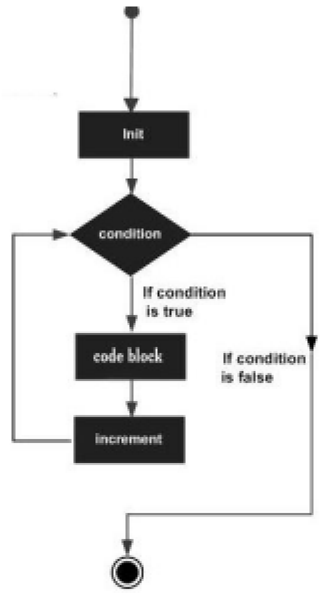
3.For Loop

The for loop in C repeats a block of code a specific number of times and uses three expressions (initialization, condition, update) separated by semicolons.

The general syntax of a for loop in C is as follows:

```
for (initialization; condition; update/increment/decrement) {
statemets;
}
```

The initialization statement is executed only once, before the loop starts. It is used to initialize variables that will be used in the loop.
The condition is checked before each iteration of the loop. If the condition is true, the code inside the loop will be executed, otherwise the loop will terminate.
The update/increment/decrement statement is executed after each iteration of the loop. It is used to update the variables used in the condition.



/*Program to print the numbers from 1 to 10 using for loop*/

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

```
{
int i;
for(i=1;i<=10;i++)
printf("%d\t", i);
printf("\n");
}
```

Output:

```
            1    2 3 4 5 6 7 8 9 10
```

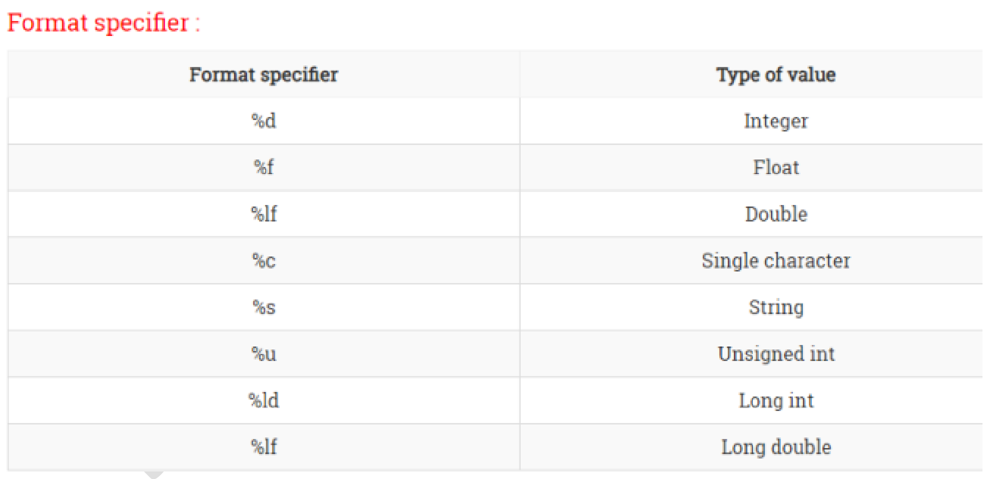
7.Explain formatted I/O functions in C.

The printf() and scanf() functions are used for input and output in C language. Both functions are inbuilt library functions, defined in stdio.h (header file).

printf() function The printf() function is used for output. It prints the given statement to the console.

The syntax of printf() function is given below:
printf("format string",argument_list);

The format string can be %d (integer), %c (character), %s (string), %f (float) etc.
scanf() function The scanf() function is used for input. It reads the input data from the console. scanf("format string",argument_list);


Format specifier :

Format specifier	Type of value
%d	Integer
%f	Float
%lf	Double
%c	Single character
%s	String
%u	Unsigned int
%ld	Long int
%lf	Long double

Program:

```
#include <stdio.h>

int main() {
    int age;
    float height;

    // Formatted input
    printf("Enter your age and height (in meters):\n");
    scanf("%d %f", &age, &height);

    // Formatted output
    printf("Your age is %d years\n", age);
    printf("Your height is %.2f meters\n", height);

    return 0;
}
```

Output:

```
Enter your age and height (in meters):
20 1.75
Your age is 20 years
Your height is 1.75 meters
```

8.Explain Decision Steps in Algorithms

Decision steps are points in an algorithm where a choice or decision is made based on some condition. Typically, this involves comparing values and branching the flow of control based on the outcome.

In programming, decision steps are implemented using:

- **if** statements
- **if-else** statements
- **else-if ladder**
- **switch-case** statements

General Structure of Decision Steps in Algorithms:

1. **Evaluate a condition**
2. **If condition is true**, do a set of instructions
3. **Else** do another set of instructions (optional)

Example: Decision step to check if a number is positive or negative

Algorithm:

```
Step 1: Start
Step 2: Read number n
Step 3: If n > 0, print "Positive"
Step 4: Else if n < 0, print "Negative"
Step 5: Else print "Zero"
Step 6: End
```

Example:

```
#include <stdio.h>

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

if (n > 0) {
printf("Positive\n");
} else if (n < 0) {
printf("Negative\n");
} else {
printf("Zero\n");
}

return 0;
}
```

Output :

```
Enter a number: 5
Positive
```

UNIT - II

1.What is top-down design in programming? Explain with an example.

BUILDING PROGRAMS FROM EXISTING INFORMATION

Top-down design breaks a complex problem into smaller sub-problems, each handled by separate functions, creating a clear hierarchical structure in C programs.

The top-down design process

The following steps outline how to apply a top-down approach with functions in C:

1. **Understand the main problem.** First, clearly define the primary goal of your program. What is the single, overarching task it needs to accomplish?

2. **Break down the problem into sub-problems.** Identify the major, high-level steps required to solve the main problem. Don't worry about the fine details yet—just define the main actions.

3. **Define functions for each sub-problem.** For each major step identified, define a function signature (return_type function_name(parameter_list);) that clearly communicates its purpose. This acts as a roadmap for your program.

4. **Refine sub-problems (stepwise refinement).** Continue breaking down each function's task into smaller sub-tasks. These smaller tasks can become calls to other, simpler functions. Repeat this process until each function is simple enough to be easily implemented.

5. **Implement the functions.** Write the code for each function, starting from the lowest-level tasks and working your way back up. This bottom-up implementation strategy ensures that functions are available to be called by higher-level functions.

6. **Integrate and test.** Combine all the functions, typically within the main() function, which serves as the top-level orchestrator. Test each component individually (unit testing) and then test how they work together.

<u>**Steps in Top-Down Design:**</u>

Understand the main problem

- Goal: Calculate the sum of two numbers and display the result.

2. Break the problem into sub-problems

- Get input from the user.
- Perform addition.
- Display the result.

3. Define functions for each sub-problem

- get_input() → To read two numbers.
- add() → To calculate the sum.
- display_result() → To show the sum.

4. Refine sub-problems (if needed)

- In this program, add() is already a simple low-level function. No further refinement needed.

5. Implement the functions

- Code each function separately:
  - Low-level: add()
  - Top-level: get_input() and display_result()
- Then integrate them in main().

6. Integrate and test

- Combine all functions in main() and run the program.
- Sample run:

```
Enter two numbers: 10 15
Sum = 25
```

Example :

```
#include <stdio.h>

// Low-level function

int add(int a, int b) {

    return a + b;

}

// Top-level functions

void get_input(int *a, int *b) {

    printf("Enter two numbers: ");

    scanf("%d %d", a, b);

}

void display_result(int sum) {

    printf("Sum = %d\n", sum);

}

int main() {

    int num1, num2, result;

    get_input(&num1, &num2);

    result = add(num1, num2);

    display_result(result);
    return 0;

}
```

Sample Run and Output:

```
Enter two numbers: 10 15
Sum = 25
```

Benefits of top-down design

- **Easy to understand:** Small functions are simpler than one big program.
- **Easy to fix:** Bugs can be found and fixed in a single function.
- **Reusable:** Functions can be used in other programs.
- **Team-friendly:** Different people can work on different functions at the same time.
- **Handles complexity:** Big problems are broken into smaller, manageable parts.

2.What are library functions? Give three examples used in C.

LIBRARY FUNCTIONS

Library functions are **predefined functions** provided by the C standard library that you can use directly in your programs. These functions perform common tasks like input/output, string handling, mathematical computations, etc., so you don't have to write them from scratch.

Key Characteristics:

1.Pre-defined:

They are already written and available as part of the C Standard Library.

2.Header Files:

Their declarations are found in header files (e.g., stdio.h, math.h, string.h), which must be included using #include directives to use the functions.

3.Efficiency and Reliability:

These functions are typically optimized for performance and have undergone extensive testing, making them reliable.

4.Reduced Development Time:

They eliminate the need to write common functionalities from scratch, saving time and effort.

Common C Standard Library Headers and Their Functions:

Category	**Header File**	**Function**	**Description**	**Example Usage**
Input/Output	<stdio.h>	printf()	Prints formatted output to console	printf("Hello %s", name);
		scanf()	Reads formatted input from console	scanf("%d", &num);
		fopen(), fclose()	Opens and closes files	FILE *fp = fopen("file.txt", "r"); fclose(fp);
String Manipulation	<string.h>	strlen()	Returns length of a string	int len = strlen(str);
		strcpy()	Copies one string to another	strcpy(dest, src);
		strcmp()	Compares two strings	if(strcmp(str1, str2) == 0) {}
Mathematical Functions	<math.h>	sqrt()	Calculates square root	double r = sqrt(16.0);
		pow()	Raises a number to a power	double p = pow(2, 3);
		sin(), cos()	Trigonometric sine and cosine functions	double s = sin(0.5);
Memory Management	<stdlib.h>	malloc(), free()	Allocates and deallocates dynamic memory	int *arr = malloc(5 * sizeof(int)); free(arr);
		atoi()	Converts string to integer	int num = atoi("123");
Character Handling	<ctype.h>	isdigit()	Checks if a character is a digit	if (isdigit(ch)) {}
		toupper(), tolower()	Converts character case	char c = toupper('a');

Example: Using Library Functions in C

1. Input/Output: printf and scanf

```
#include <stdio.h>

int main() {
int num;
printf("Enter a number: ");
scanf("%d", &num);
printf("You entered: %d\n", num);
return 0;
}
```

Sample Run:

```
Enter a number: 42
You entered: 42
```

2. String Manipulation: strlen and strcpy

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

int main() {
char src[] = "Hello";
char dest[20];
strcpy(dest, src);
printf("Copied string: %s\n", dest);
printf("Length of string: %d\n", (int)strlen(dest));
return 0;
}
```

Output:

```
Copied string: Hello
Length of string: 5
```

3. Mathematical Functions: sqrt and pow

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

int main() {
double x = sqrt(9.0);
double y = pow(2, 3);
printf("Square root of 9 is %.2f\n", x);
printf("2 raised to power 3 is %.2f\n", y);
return 0;
}
```

Output:

```
Square root of 9 is 3.00
2 raised to power 3 is 8.00
```

4. Memory Management: malloc and free

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

int main() {
int *arr = (int*) malloc(3 * sizeof(int));
if (arr == NULL) {
printf("Memory allocation failed\n");
return 1;
}
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
printf("Array elements: %d %d %d\n", arr[0], arr[1], arr[2]);
free(arr);
return 0;
}
```

Output:

```
Array elements: 10 20 30
```

5. Character Handling: isdigit, toupper, and tolower

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

int main() {
char ch = 'a';
if (isdigit(ch)) {
printf("%c is a digit\n", ch);
} else {
printf("%c is not a digit\n", ch);
}
printf("Uppercase of %c is %c\n", ch, toupper(ch));
printf("Lowercase of %c is %c\n", 'B', tolower('B'));
return 0;
}
```

Output:

```
a is not a digit
Uppercase of a is A
Lowercase of B is b
```

3.Explain functions with input arguments and functions without arguments.

Functions with Input Arguments in C

A **function with input arguments** takes one or more values (parameters) when called. These arguments are used inside the function to perform tasks based on the input.

Syntax

```
return_type function_name(parameter_type1 param1, parameter_type2 param2, ...) {
// function body
}
```

- The function can take any number of parameters.
- Each parameter has a **type** and a **name**.
- You pass actual values (arguments) when calling the function.

Example 1: Function with two integer arguments and no return value

A **function with two integer arguments and no return value**:

- Takes **two integer inputs** (arguments) when called.
- Performs a task using these two integers inside the function.
- Does **not return any value** to the caller (its return type is void).

```
#include <stdio.h>
void printSum(int a, int b) {
int sum = a + b;
printf("Sum is %d\n", sum);
}

int main() {
printSum(5, 7);  // Pass 5 and 7 as arguments
return 0;
}
```

Output:

```
Sum is 12
```

Example 2: Function with arguments and return value

A **function with arguments and return value** is a block of code that:

- **Accepts one or more input parameters (arguments)** when called.
- **Performs operations** using these input arguments inside the function body.
- **Returns a value** of a specified data type back to the part of the program that called the function.

```
#include <stdio.h>
int multiply(int x, int y) {
return x * y;
}

int main() {
int result = multiply(4, 6);  // Pass 4 and 6 as arguments
printf("Product is %d\n", result);
return 0;
}
```

Output:

```
Product is 24
```

Functions without Arguments in C

A **function without arguments** is a function that does not take any parameters when called. It performs a specific task but does not require any input values from the caller.

Syntax

```
return_type function_name(void) {
// function body
}
```

- void inside the parentheses means the function takes no arguments.
- The function can return a value or be void if it returns nothing.

Example 1: Function without arguments and without return value

A **function without arguments and without return value**:

- **Does not take any input parameters** (no arguments).
- **Performs a specific task** inside its body.

- **Does not return any value** to the caller (its return type is void).

```
#include <stdio.h>

void greet(void) {
printf("Hello, welcome!\n");
}

int main() {
greet();  // Call function without arguments
return 0;
}
```

Output:

```
Hello, welcome!
```

Example 2: Function without arguments but returns a value

A **function without arguments but returns a value**:

- **Takes no input parameters** (no arguments).
- Performs some operation or calculation inside its body.
- **Returns a value** of a specific type back to the caller.

```
#include <stdio.h>
int getNumber(void) {
return 42;
}

int main() {
int num = getNumber();
printf("Number is %d\n", num);
return 0;
}
```

Output:

```
Number is 42
```

4.Explain pointers and the indirection (*) operator with an example.

POINTER

A **pointer** is a variable whose **value is the memory address** of another variable. Instead of holding the actual data, it **points** to where the data is stored in memory.

How to Declare and Use Pointers

Declaration

```
int *p;
```

- p is declared as a pointer to an integer (int).
- The * symbol here means “pointer to”.

Assigning Addresses

```
int a = 10;
p = &a;
```

- The & operator means **"address of"**.
- So, &a is the memory address where variable a is stored.
- Now, p holds the address of a.

Accessing Values via Pointers (Dereferencing)

```
printf("%d\n", *p);
```

- The * operator here means **"value pointed to by"** (dereferencing).
- *p accesses the value stored at the memory location stored in p.
- So, *p gives the value of a (which is 10).

Dereference (Indirection) Operator (*)

The * operator is used to **access or modify** the value stored at the address the pointer points to.

```
int x = 10;
int *p = &x;

printf("%d\n", *p);  // prints 10 (value at address p)
*p = 20;             // changes x to 20
printf("%d\n", x);   // prints 20
```

Example :

```
#include <stdio.h>
int main() {
int a = 10;
int *p;     // Declare pointer p to int
p = &a;     // Store address of a in p
printf("Value of a: %d\n", a);          // Output: 10
printf("Address of a: %p\n", &a);       // Output: address of a (memory location)
printf("Value stored in p: %p\n", p);   // Output: same address of a
printf("Value pointed to by p: %d\n", *p); // Output: 10 (value at address p)

return 0;
}
```

Output

```
Value of a: 10
Address of a: 0x7ffee7c4b6ac
Value stored in p: 0x7ffee7c4b6ac
Value pointed to by p: 10
```

5.How do you pass input & output parameters to a function using pointers?

A function in C can be called multiple times with different input values, and it can return output each time. This is useful when you want to reuse the same logic for different inputs.

How it works:

- You define a function that takes input parameters and returns an output.
- You call this function multiple times with different arguments.
- Each call processes the inputs and returns a result.

Example:

```
#include <stdio.h>

// Function prototype
int multiply(int a, int b);

int main() {
int result1, result2, result3;

// Multiple calls to the function with different inputs
result1 = multiply(2, 3);  // 6
result2 = multiply(5, 4);  // 20
result3 = multiply(7, 8);  // 56

printf("2 x 3 = %d\n", result1);
printf("5 x 4 = %d\n", result2);
printf("7 x 8 = %d\n", result3);
return 0;
}

// Function definition
int multiply(int a, int b) {
return a * b;
}
```

Output:

```
2x 3 = 6
5x 4 = 20
7x 8 = 56
```

6.What is the scope of a variable in C? Differentiate between local and global variables.

The **scope of a variable** in C is the block or the region in the program where a variable is declared, defined, and used. Outside this region, we cannot access the variable, and it is treated as an undeclared identifier.

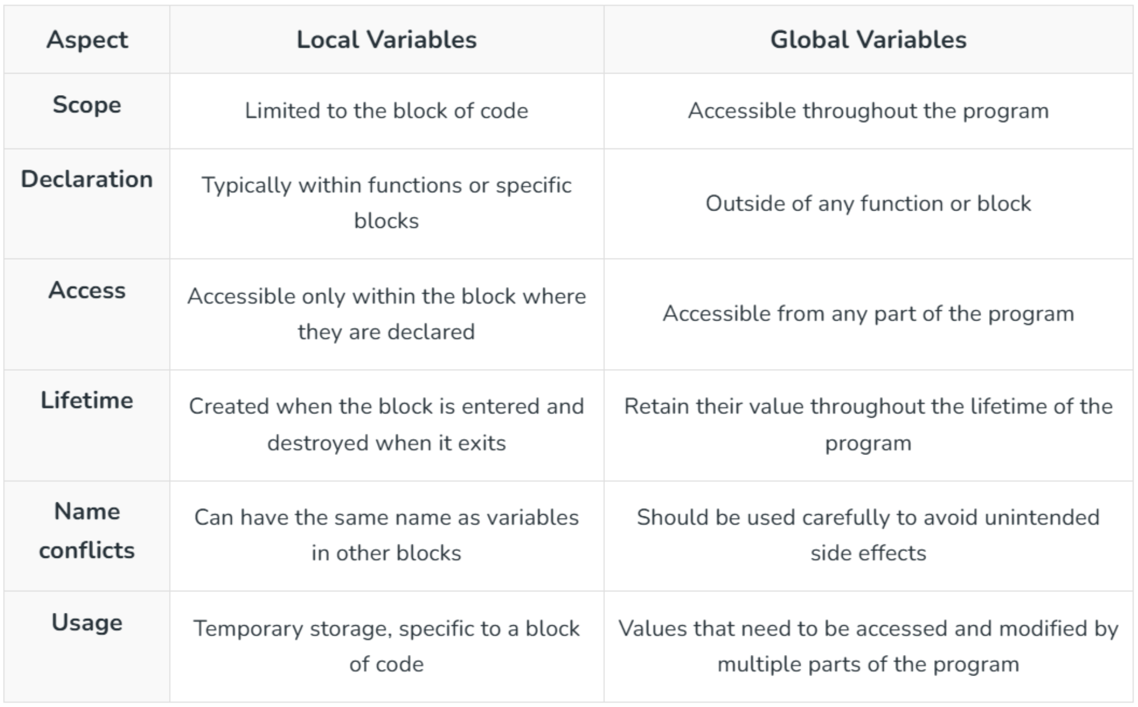
- The scope of an identifier is the part of the program where the identifier may directly be accessible.
- All variables are lexically(or statically) scoped in C which means the scope is defined at the compiler time and not dependent on the caller of the function.

```
#include <stdio.h>

int globalVar = 50;   // Global variable

void display() {

    int localVar = 20;   // Local variable

    printf("Inside display function:\n");

    printf("Local Variable = %d\n", localVar);

    printf("Global Variable = %d\n", globalVar);

}

int main() {
    display();

    printf("\nInside main function:\n");

    printf("Global Variable = %d\n", globalVar);

    return 0;

}
```

OUTPUT:

```
Inside display function:

Local Variable = 20

Global Variable = 50

Inside main function:

Global Variable = 50
```

Difference between local and global variables



Aspect	Local Variables	Global Variables
Scope	Limited to the block of code	Accessible throughout the program
Declaration	Typically within functions or specific blocks	Outside of any function or block
Access	Accessible only within the block where they are declared	Accessible from any part of the program
Lifetime	Created when the block is entered and destroyed when it exits	Retain their value throughout the lifetime of the program
Name conflicts	Can have the same name as variables in other blocks	Should be used carefully to avoid unintended side effects
Usage	Temporary storage, specific to a block of code	Values that need to be accessed and modified by multiple parts of the program

7.Explain formal output parameters as actual arguments with an example.

In **C programming**, the concept of **"output parameters as actual arguments"** refers to using **call by reference** to allow a function to **modify the value of a variable in the calling function**. This is done by passing the **memory address** of the variable to the function, which then operates on the data at that address.

Key Concepts

- **Formal Parameters**:
  These are the variables declared in the function definition's parameter list. They serve as placeholders for the values or addresses passed to the function.
- **Actual Arguments**:
  These are the values or variables passed to a function when it is called.

Output Parameters as Actual Arguments (Call by Reference)

When you want a function to **modify** a variable from the calling function (i.e., to use it as an "output"), you pass the variable's **address** using the & operator.
The function's formal parameter should then be a **pointer**, which can access and modify the value stored at that memory address.

Example in C

```
#include <stdio.h>

// Function to increment a value using call by reference
void increment(int *num) { // 'num' is a formal parameter, a pointer to an int
(*num)++; // Dereference the pointer to modify the value at the address
}

int main() {
int value = 10;
printf("Before increment: %d\n", value);

increment(&value); // Pass the address of 'value' as an actual argument

printf("After increment: %d\n", value);
return 0;
}
```

Result:

```
Before increment: 10
```

```
After increment: 11
```

Explanation:

- **void increment(int *num)**:
  The increment function is defined to take a pointer to an integer. This pointer is the **formal parameter**.
- **increment(&value);**:
  In main, the **address** of value is passed as the **actual argument** using &value.
- **(*num)++**:
  Inside the increment function, num points to the memory location of value.
  The expression *num accesses the actual value at that address, and (*num)++ increments it.
  This change is reflected in main because it modifies the original variable.

8.Write a program using a function to swap two numbers using pointers.

Program:

```
#include <stdio.h>

// Function to swap two numbers using pointers

void swap(int *a, int *b) {

    int temp;

    temp = *a;

    *a = *b;

    *b = temp;

}

int main() {

    int x, y;

    printf("Enter two numbers:\n");

    scanf("%d %d", &x, &y);

    printf("Before swapping:\n");
    printf("x = %d, y = %d\n", x, y);

    swap(&x, &y);   // Function call using addresses

    printf("After swapping:\n");

    printf("x = %d, y = %d\n", x, y);

    return 0;

}
```

Output:

```
Enter two numbers:

10 20

Before swapping:

x = 10, y = 20

After swapping:

x = 20, y = 10
```

9.Explain modular programming and its advantages.

Modular programming **is a software design technique that involves** dividing a program into separate, independent modules or files**, where each module is responsible for a specific part of the program's functionality. This approach helps improve** code organization, readability, maintainability, and makes it easier for multiple programmers to collaborate **on the same project.**

How to do modular programming in C

- Use **header files** (.h) to declare function prototypes.
- Use **source files** (.c) to define functions.
- The main file (main.c) includes headers and calls functions.

Example:

File: add.h

```
#ifndef ADD_H
#define ADD_H

int add(int a, int b);

#endif
```

File: add.c

```
#include "add.h"

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

File: main.c

```
#include <stdio.h>
#include "add.h"

int main() {
int x = 5, y = 10;
printf("Sum: %d\n", add(x, y));
return 0;
}
```

Compile the program

```
gcc main.c add.c -o program
./program
```

Output:

```
Sum: 15
```

Another example Combining Pointers & Modular Programming

operations.h

```
#ifndef OPERATIONS_H
#define OPERATIONS_H

void swap(int *a, int *b);
int add(int a, int b);

#endif
```

operations.c

```
#include "operations.h"

void swap(int *a, int *b) {
int temp = *a;
*a = *b;
*b = temp;
}

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

main.c

```
#include <stdio.h>
#include "operations.h"

int main() {
int x = 10, y = 20;

printf("Before swap: x = %d, y = %d\n", x, y);
swap(&x, &y);
printf("After swap: x = %d, y = %d\n", x, y);

int sum = add(x, y);
printf("Sum of x and y = %d\n", sum);

return 0;
}
```

Compile and run:

```
gcc main.c operations.c -o program
./program
```

Output:

```
Before swap: x = 10, y = 20
After swap: x = 20, y = 10
Sum of x and y = 30
```

UNIT – III

1.Explain array subscripts and sequential access using a for loop.

An **array subscript** is an integer value used inside square brackets [] to specify the position of an element within an array. It tells the program which particular element of the array to access or modify.

In most programming languages like C, array subscripts start at **0** (zero-based indexing), meaning the first element of the array is accessed with subscript 0, the second element with subscript 1, and so on.

Example:

```
int numbers[5] = {10, 20, 30, 40, 50};
int x = numbers[2];  // Here, 2 is the subscript, accessing the 3rd element (30)
```

Key Points About Array Subscripts in C

Zero-based indexing
The first element of an array is accessed with subscript 0.
The last element is accessed with subscript array_size - 1.

Subscript must be an integer
Typically an int or a variable holding an integer value.
Using non-integer values is invalid.

Subscript must be within bounds
Valid subscripts range from 0 to array_size - 1.
Accessing outside this range leads to **undefined behavior** (can crash the program or corrupt data).

Subscripts can be variables

```
int i = 3;
int value = numbers[i];  // accesses the 4th element
```

Subscripts can be expressions

```
int value = numbers[1 + 2];  // equivalent to numbers[3]
```

Example Demonstrating Subscripts:

```
#include <stdio.h>

int main() {
int arr[4] = {5, 10, 15, 20};

for (int i = 0; i < 4; i++) {
printf("Element at index %d is %d\n", i, arr[i]);
}

return 0;
}
```

Output:

```
Element at index 0 is 5
Element at index 1 is 10
Element at index 2 is 15
Element at index 3 is 20
```

USING FOR LOOPS FOR SEQUENTIAL ACCESS

A **for loop** is commonly used to **sequentially access** each element of an array. This means you start from the first element (index 0) and move one by one to the last element (index array_size - 1).

Why Use a For Loop?

- It automates the process of accessing each element.
- It avoids repetitive code.
- It is easy to read and maintain.

General Syntax

```
for (int i = 0; i < array_size; i++) {
// Access array element using subscript i
// array[i]
}
```

- i is the loop variable acting as the array subscript.
- The loop runs from 0 to array_size - 1, covering all elements.

Example: Sequentially Printing Array Elements

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

```
int main() {
int numbers[5] = {10, 20, 30, 40, 50};

for (int i = 0; i < 5; i++) {
printf("Element at index %d is %d\n", i, numbers[i]);
}

return 0;
}
```

Output:

```
Element at index 0 is 10
Element at index 1 is 20
Element at index 2 is 30
Element at index 3 is 40
Element at index 4 is 50
```

2.How do you declare and initialize an array in C?

An **array** is a collection of elements of the same data type stored in **contiguous memory locations**. It allows you to store multiple values under a single variable name, accessible by an **index**.

Declaring Arrays in C

When you declare an array, you tell the compiler:

- The **type** of elements the array will hold (e.g., int, char, float).
- The **name** of the array.
- The **size** (number of elements) of the array.

Syntax

data_type array_name[size];

data_type:

This specifies the type of data that the array will store (e.g., int, float, char). All elements in an array must be of the same data type.

array_name:

This is the identifier you choose for your array. It must follow C's variable naming rules.

size:

This is a positive integer constant that determines the number of elements the array can store. It is enclosed in square brackets [].

Example:

int numbers[5];

This declares an integer array called numbers that can hold **5 integers**.

- The array size must be a constant value.
- The index ranges from 0 to array_size - 1.

3. Initializing Arrays at Declaration

You can also initialize the array at the time of declaration:

int numbers[5] = {10, 20, 30, 40, 50};

- If you don't specify all values, the remaining elements are set to 0.
- You can omit the size if you provide initialization values:

int numbers[] = {10, 20, 30};

Here, the array size is automatically set to 3.

4. Referencing (Accessing) Array Elements

Each element in the array can be accessed using its **index**.

Syntax:

array_name[index]

Example:

```
int firstNumber = numbers[0];  // Access first element
numbers[2] = 100;              // Set the third element to 100
```

- Array indices start at **0**.
- So, for numbers[5], valid indices are 0, 1, 2, 3, 4.

Example

```
#include <stdio.h>

int main() {
int numbers[5] = {10, 20, 30, 40, 50};
// Access elements
printf("First number: %d\n", numbers[0]);
printf("Third number: %d\n", numbers[2]);

// Modify element
numbers[4] = 100;

printf("Modified fifth number: %d\n", numbers[4]);

return 0;
}
```

Output:

```
First number: 10
Third number: 30
Modified fifth number: 100
```

3.How can array elements be passed as arguments to functions?

In C, individual elements of an array can be passed to a function as arguments just like any other variable of the corresponding data type. Accessing an element via array[index] gives you the value stored at that position, which can be passed directly to functions.

Example Code:

```
#include <stdio.h>

// Function to add two integers
int addNumbers(int a, int b) {
return a + b;
}

int main() {
int numbers[] = {10, 20, 30, 40, 50};
int result;

// Passing individual array elements as arguments to the function
result = addNumbers(numbers[0], numbers[1]);
printf("Sum of numbers[0] and numbers[1]: %d\n", result);

result = addNumbers(numbers[2], numbers[4]);
printf("Sum of numbers[2] and numbers[4]: %d\n", result);

return 0;
}
```

Explanation:

- **Function addNumbers**: Takes two integer parameters a and b, and returns their sum.
- **In main**:
- The integer array numbers is declared and initialized with 5 elements.
- The first call addNumbers(numbers[0], numbers[1]) passes numbers[0] (which is 10) and numbers[1] (which is 20).
- The second call addNumbers(numbers[2], numbers[4]) passes numbers[2] (which is 30) and numbers[4] (which is 50).
- The results are stored in the variable result and printed.
- This demonstrates that numbers[index] accesses the value at that index, which can then be passed to a function like any other integer variable.

4.Write a program to search for an element in an array.

Searching an Array

Searching in an array means looking for a specific element (value) within the array and determining its position (index). If the element is found, the position is returned; otherwise, it indicates that the element is not present.

Common Searching Algorithms:

a) Linear Search

- Checks every element from the start to the end until the target is found.
- Works on **both sorted and unsorted** arrays.
- Time complexity: O(n).

Example:

```
int linearSearch(int arr[], int size, int target) {
for(int i = 0; i < size; i++) {
if (arr[i] == target)
return i;  // Return index if found
}
return -1;  // Not found
}
```

b) Binary Search (only for sorted arrays)

- Divides the array in half repeatedly to find the target.
- Much faster than linear search for large sorted arrays.

- Time complexity: O(log n).

Example:

```
int binarySearch(int arr[], int size, int target) {
int low = 0, high = size - 1;

while(low <= high) {
int mid = low + (high - low) / 2;

if (arr[mid] == target)
return mid;
else if (arr[mid] < target)
low = mid + 1;
else
high = mid - 1;
}
return -1;  // Not found
}
```

5.Explain sorting. Discuss different sorting techniques in detail.

Sorting is the process of rearranging the elements of an array into a specific order (usually ascending or descending).

Common Sorting Algorithms:

a) Bubble Sort

- Repeatedly swaps adjacent elements if they are in the wrong order.
- Simple but inefficient for large datasets.
- Time complexity: O(n²).

```
void bubbleSort(int arr[], int size) {
for(int i = 0; i < size - 1; i++) {
for(int j = 0; j < size - 1 - i; j++) {
if(arr[j] > arr[j + 1]) {
int temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
}
}
}
}
```

b) Selection Sort

- Selects the minimum element from the unsorted part and swaps it with the first unsorted element.
- Time complexity: $O(n^2)$.

```
void selectionSort(int arr[], int size) {
for(int i = 0; i < size - 1; i++) {
int minIndex = i;
for(int j = i + 1; j < size; j++) {
if(arr[j] < arr[minIndex])
minIndex = j;
}
int temp = arr[minIndex];
arr[minIndex] = arr[i];
arr[i] = temp;
}
}
```

c) Insertion Sort

- Builds the sorted array one element at a time by inserting elements into their correct position.
- Time complexity: $O(n^2)$.

```
void insertionSort(int arr[], int size) {
for(int i = 1; i < size; i++) {
int key = arr[i];
int j = i - 1;
while(j >= 0 && arr[j] > key) {
arr[j + 1] = arr[j];
         j--;
    }
    arr[j + 1] = key;
  }
}
```

6.What are parallel arrays? Give an example.

Parallel Arrays in C

Parallel arrays are multiple arrays where related data is stored at the same index in each array. Instead of using a single array of structs, related attributes are stored in separate arrays but aligned by index.

When to Use:

- When you want to keep related data in separate arrays.
- Sometimes used for simplicity or when struct usage is not desired.

Example:

Imagine storing information about students — their IDs, names, and grades — using parallel arrays.

```
#include <stdio.h>

int main() {
int studentIDs[] = {101, 102, 103};
char *studentNames[] = {"Alice", "Bob", "Charlie"};
float studentGrades[] = {85.5, 90.0, 78.0};
int size = 3;

for (int i = 0; i < size; i++) {
printf("Student ID: %d, Name: %s, Grade: %.2f\n",
studentIDs[i], studentNames[i], studentGrades[i]);
}

return 0;
}
```

OUTPUT

```
Student ID: 101, Name: Alice, Grade: 85.50
Student ID: 102, Name: Bob, Grade: 90.00
Student ID: 103, Name: Charlie, Grade: 78.00
```

7.Explain multidimensional arrays with an example of a 2D array.

A **multidimensional array** is basically an array of arrays. The most common example is a **2D array**, which you can think of as a table or matrix (rows and columns). C also supports higher dimensions like 3D arrays, but 2D arrays are most common.

Declaration and Initialization

2D Array Declaration:

```
type arrayName[rows][columns];
```

Example:

```
int matrix[3][4];  // 3 rows, 4 columns
```

Initialization Example:

```
int matrix[2][3] = {
{1, 2, 3},
{4, 5, 6}
};
```

or

```
int matrix[2][3] = {1, 2, 3, 4, 5, 6};  // Row-major order
```

Accessing Elements

Access elements by specifying both row and column indices:

```
int x = matrix[0][2];  // Access element in first row, third column (value 3)
matrix[1][0] = 10;     // Change element in second row, first column to 10
```

Example Program: Print a 2D Array

```
#include <stdio.h>

int main() {
int matrix[3][3] = {
{1, 2, 3},
{4, 5, 6},
{7, 8, 9}
};

// Print the 2D array
for(int i = 0; i < 3; i++) {        // Loop over rows
for(int j = 0; j < 3; j++) {    // Loop over columns
printf("%d ", matrix[i][j]);
}
printf("\n");
}

return 0;
}
```

Output:

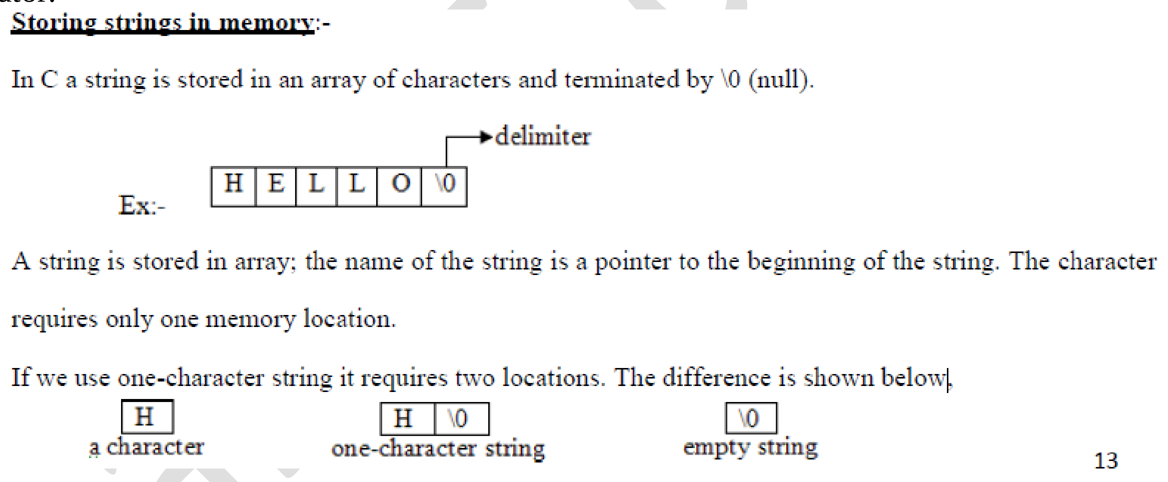
```
            1   2 3
4 5 6
7 8 9
```

8.How are strings stored in C? Explain null-termination.

In C language a string is group of characters (or) array of characters, which is terminated by delimiter \0 (null). Thus, C uses variable-length delimited strings in programs.

Declaring Strings:-
C does not support string as a data type. It allows us to represent strings as character arrays. In C, a string variable is any valid C variable name and is always declared as an array of characters.
Syntax:- char string_name[size];
The size determines the number of characters in the string name.
Ex:- char city[10]; char name[30];

Initializing strings:-
There are several methods to initialize values for string variables.
Ex:- char city[8]="NEWYORK";
char city[8]={„N“,“E“,“W“,“Y“,“O“,“R“,“K“,“\0“};
The string city size is 8 but it contains 7 characters and one character space is for NULL terminator.



Storing strings in memory:-

In C a string is stored in an array of characters and terminated by \0 (null).

delimiter
Ex:-
H
E
L
L
O
\0

A string is stored in array; the name of the string is a pointer to the beginning of the string. The character requires only one memory location.

If we use one-character string it requires two locations. The difference is shown below.

H
a character
H
\0
one-character string
\0
empty string
13

9.Explain string library functions?

C supports a number of string handling functions defined in the header file <string.h>. These built-in functions perform various operations on strings.

(i) strlen()

- **Purpose:** Finds the length of a string (excluding the null terminator \0).
- **Syntax:**
- int strlen(const char *s);
- **Example:**

```
#include <stdio.h>
#include <string.h>
int main() {
char str[20];
int length;

printf("Enter any string: ");
gets(str);  // Note: unsafe, use fgets in real programs
length = strlen(str);
printf("The length of the string is: %d\n", length);
return 0;
}
```

Output:

```
Enter any string: C PROGRAMMING
The length of the string is: 13
```

(ii) strcpy()

- **Purpose:** Copies one string into another.
- **Syntax:**
- char *strcpy(char *dest, const char *src);
- **Example:**

```
#include <stdio.h>
#include <string.h>
int main() {
char source[20], target[20];
printf("Enter the string: ");
gets(source);
strcpy(target, source);
printf("The target string is: %s\n", target);
return 0;
}
```

Output:

```
Enter the string: WELCOME
The target string is: WELCOME
```

(iii) strcmp()

- **Purpose:** Compares two strings for equality.
- **Syntax:**
- int strcmp(const char *s1, const char *s2);
- Returns 0 if strings are equal, a negative value if s1 < s2, positive if s1 > s2.
- **Example:**

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

int main() {
char str1[20], str2[20];

printf("Enter the first string: ");
gets(str1);
printf("Enter the second string: ");
gets(str2);

if (strcmp(str1, str2) == 0)
printf("Both strings are equal\n");
else
printf("Both strings are not equal\n");

return 0;
}
```

Output:

```
Enter the first string: welcome
Enter the second string: welcome
Both strings are equal
```

(iv) strcat()

- **Purpose:** Concatenates (appends) one string to another.
- **Syntax:**
- char *strcat(char *dest, const char *src);
- **Example:**

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

int main() {
char source[20], target[40];

printf("Enter the target string: ");
gets(target);
printf("Enter the source string: ");
gets(source);

strcat(target, source);

printf("Now target string is: %s\n", target);
return 0;
}
```

Output:

```
Enter the target string: data
Enter the source string: base
Now target string is: database
```

(v) strstr()

- **Purpose:** Finds the first occurrence of a substring in a string.
- **Syntax:**
- char *strstr(const char *haystack, const char *needle);
- **Example:**

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

int main() {
char str[100] = "this is javatpoint with c and java";
char *sub;

sub = strstr(str, "java");

if (sub != NULL)
printf("Substring is: %s\n", sub);
else
printf("Substring not found.\n");

return 0;
}
```

Output:

```
Substring is: java tpoint with c and java
```

Additional String Functions (Non-Standard but common in some compilers)

strrev() - Reverse string

- **Purpose:** Reverses the given string.
- **Syntax:**
- char *strrev(char *str);
- **Example:**

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

int main() {
char str[50];
printf("Enter string: ");
gets(str);

strrev(str);
printf("Reversed string is: %s\n", str);
return 0;
}
```

Output:

```
Enter string: welcome to hyderabad
Reversed string is: dabaredyh ot emoclew
```

Note: *strrev()* is not part of the standard C library, so it may not be available in all environments.

strupr() - Convert string to uppercase

- **Purpose:** Converts all characters of a string to uppercase.
- **Syntax:**
- char *strupr(char *str);
- **Example:**

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

int main() {
char str[50];
printf("Enter string: ");
gets(str);

printf("Uppercase string: %s\n", strupr(str));
return 0;
}
```

Note: strupr() is also not part of the standard C library.

strlwr() - Convert string to lowercase

- **Purpose:** Converts all characters of a string to lowercase.
- **Syntax:**
- char *strlwr(char *str);
- **Example:**

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

int main() {
char str[50];
printf("Enter string: ");
gets(str);

printf("Lowercase string: %s\n", strlwr(str));
return 0;
}
```

10.How can you create an array of pointers to strings in C?

An **array of pointers** is simply an array whose **elements are pointers** instead of normal variables.

Example:

```
int *arr[5];
```

Here:

- arr is an array with **5 elements**.
- Each element (arr[0], arr[1], … arr[4]) is a **pointer to int**.

Use Arrays of Pointers

They are useful when:

- You want to store **addresses** of multiple variables.
- You want to manage a list of **strings** (common in C).
- You want to handle **dynamic data** efficiently (e.g., arrays of varying lengths).

Example — Array of Pointers to Integers

```
#include <stdio.h>
int main() {
int a = 10, b = 20, c = 30;
int *arr[3]; // Array of 3 int pointers

arr[0] = &a;
arr[1] = &b;
arr[2] = &c;

for (int i = 0; i < 3; i++) {
printf("Value of arr[%d] = %d\n", i, *arr[i]);
}

return 0;
}
```

Output:

```
Value of arr[0] = 10
Value of arr[1] = 20
Value of arr[2] = 30
```

UNIT – IV

1.What is recursion? Explain recursion with an example of a factorial function

Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied. In programming languages, if a program allows us to call a function inside the same function, then it is called a recursive call to the function

While using recursion, programmers need to be careful to define an exit condition from the recursive function; otherwise it will go into an infinite loop.

Syntax:

```
return_type fuction_name(parameter list)
{ local variable declarations;
stmt 1;
simt 2;
…………………
```

If(condition) stop condition/

Return;

else

function name(); recursive call ie. calling itself"/

return statement;

}

Types of Recursion:

1. Direct Recursion: A function is said to be direct recursive if it calls itself directly

2. Indirect Recursion: A function is said to be indirect recursive if it calls another function and this new function calls the first calling function again

C Program: Recursive Factorial Function

```
#include <stdio.h>

// Recursive function to calculate factorial
long long factorial(int n) {
// Base Case: If n is 0, return 1
if (n == 0) {
return 1;
}
// Recursive Case: n * factorial(n-1)
else {
return n * factorial(n - 1);
}
}

int main() {
int num;
printf("Enter a non-negative integer: ");
scanf("%d", &num);

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

return 0;
}
```

OUTPUT :

```
Enter a non-negative integer: 5
```

Factorial of 5 = 120

The Nature of Recursion

Recursion in programming is a technique where a **function calls itself** to solve a problem. It helps break down complex problems into **smaller, simpler sub-problems** that are similar in structure to the original problem.

Core Principles of Recursion

1.Base Case(s)

- A base case is a simple case of the problem that can be solved **without further recursion**.
- It provides the **termination condition**, preventing infinite recursive calls.
- Every recursive function **must** have at least one base case.

2.Recursive Step

- This is the part of the function where the problem is **reduced to a smaller version** of itself.
- The function calls itself with this smaller input.
- Each recursive call gets closer to the **base case**.

2.How do you trace a recursive function? Explain with a flow diagram.

Tracing a Recursive Function in C

Tracing means observing:

- How recursive calls **build up** on the call stack
- How values are **returned** as the function unwinds

Let's trace **Fibonacci using recursion** in a similar step-by-step manner.

Example Program: Fibonacci in C

```
#include <stdio.h>

int fibonacci(int n) {
// Base cases
if (n == 0)
return 0;
else if (n == 1)
return 1;
// Recursive step
else
return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
int n = 5;
printf("Fibonacci of %d is %d\n", n, fibonacci(n));
return 0;
}
```

Tracing fibonacci(5)

Step-by-step (Function Calls Going Down)

1. fibonacci(5)
    - n = 5 → not base case
    - returns fibonacci(4) + fibonacci(3)
2. fibonacci(4)
    - n = 4
    - returns fibonacci(3) + fibonacci(2)
3. fibonacci(3)
    - n = 3
    - returns fibonacci(2) + fibonacci(1)
4. fibonacci(2)
    - n = 2
    - returns fibonacci(1) + fibonacci(0)
5. fibonacci(1) → base case → returns 1
6. fibonacci(0) → base case → returns 0
7. fibonacci(2) receives 1 + 0 = 1 → returns 1
8. fibonacci(1) → base case → returns 1
9. fibonacci(3) receives 1 + 1 = 2 → returns 2
10. fibonacci(2) → calls fibonacci(1) + fibonacci(0) → 1 + 0 = 1
11. fibonacci(4) receives 2 + 1 = 3 → returns 3
12. fibonacci(3) → calls fibonacci(2) + fibonacci(1) → 1 + 1 = 2
13. fibonacci(5) receives 3 + 2 = 5 → returns 5

Returning Values (Unwinding the Stack)

Function Call	Value Returned
fibonacci(1)	1
fibonacci(0)	0
fibonacci(2)	1
fibonacci(1)	1
fibonacci(3)	2

Function Call	**Value Returned**
fibonacci(2)	1
fibonacci(4)	3
fibonacci(3)	2
fibonacci(5)	5

The main() prints:

Fibonacci of 5 is 5

3.Write a recursive function to calculate the Fibonacci series.

```
#include <stdio.h>

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 the number of terms: ");

    scanf("%d", &n);

    printf("Fibonacci series:\n");

    for (i = 0; i < n; i++) {

        printf("%d ", fibonacci(i));

    }
    return 0;

}
```

OUTPUT:

```
Enter the number of terms: 6

Fibonacci series:

              0    1 1 2 3 5
```

4.Explain the use of arrays and strings in recursive functions.

Recursive functions in C can effectively process arrays and strings by passing pointers as parameters. Each recursive call handles a smaller portion of the data until reaching a clearly-defined **base case**, preventing infinite recursion.

Processing Arrays Recursively

In C, arrays are typically passed as pointers to their first elements. To traverse or process them recursively, you usually pass:

- the array itself
- the current index
- the size of the array

Example: Printing Array Elements Recursively

```
#include <stdio.h>

void printArray(int arr[], int n) {
if (n == 0) // base case
return;
printArray(arr, n - 1); // recursive call
printf("%d ", arr[n - 1]); // print element
}

int main() {
int arr[] = {1, 2, 3, 4, 5};
int size = 5;
printf("Array elements: ");
printArray(arr, size);
printf("\n");
return 0;
}
```

Output:

```
Array elements: 1 2 3 4 5
```

Processing Strings Recursively

Strings in C are null-terminated character arrays ('\0'). Because of this, recursion on strings is naturally expressed by advancing the pointer to the next character.

Example: Printing a String Recursively

```
#include <stdio.h>

void printString(char *str) {

if (*str == '\0') // base case

return;

printf("%c", *str); // print current character

printString(str + 1); // recursive call

}

int main() {

char str[] = "Hi";

printf("String: ");

printString(str);

printf("\n");

return 0;

}
```

Output:

```
String: Hi
```

5.What are structures in C? Give an example.

A **structure** in C is a user-defined data type that allows you to combine **different types of variables** (like `int`, `float`, `char`, etc.) under a single name. It is useful for representing a "record" that has multiple attributes.

Syntax of a Structure

```
struct structure_name {
data_type member1;
data_type member2;
...
};
```

- `struct` → keyword to define a structure
- `structure_name` → name of the structure
- `member1, member2` → variables inside the structure (called **members** or **fields**)

Declaring Structure Variables

You can declare variables in three ways:

1. **At the time of definition:**

```
struct Student {
int id;
char name[50];
float marks;
} s1, s2;  // declare variables immediately
```

2. **After definition:**

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

3. **Using `typedef`** (to avoid writing `struct` every time):

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

Student s1, s2;  // no need to write 'struct'
```

Accessing Structure Members

- **Dot operator** `.` for normal structure variables:

```
s1.id = 101;
```

- **Arrow operator** **->** for pointers to structures:

```
struct Student *ptr = &s1;
ptr->marks = 98.0;
```

Example

```
#include <stdio.h>

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

int main() {
struct Student s1;

// Assign values
s1.id = 101;
strcpy(s1.name, "Alice"); // <string.h> is needed for strcpy
s1.marks = 95.5;

// Print values
printf("ID: %d\n", s1.id);
printf("Name: %s\n", s1.name);
printf("Marks: %.2f\n", s1.marks);

return 0;
}
```

Explanation:

- `struct Student` defines a new structure type.
- `s1` is a variable of type `struct Student`.
- We access structure members using the **dot operator** `.`.

6.How can structure data be passed as input/output parameters to?

A **structure** is like a container that holds **different types of data** together.

```
struct Student {
int id;
char name[20];
float marks;
};
```

Passing Structure by Value

Passing a structure **by value** means the function receives a **copy** of the structure. Any changes inside the function **do not affect** the original structure.

Example:

```
#include <stdio.h>

struct Student {
int id;
char name[20];
float marks;
};

void printStudent(struct Student s) {
printf("ID: %d\n", s.id);
printf("Name: %s\n", s.name);
printf("Marks: %.2f\n", s.marks);

s.marks = 0; // Change here won't affect original
}

int main() {
struct Student s1 = {1, "Bob", 80.5};

printStudent(s1);

printf("Original Marks: %.2f\n", s1.marks); // Still 80.5

return 0;
}
```

OUTPUT

```
ID: 1
Name: Bob
Marks: 80.50
Original Marks: 80.50
```

Passing Structure by Reference (Pointer)

Passing a structure **by reference** means the function receives the **address** of the structure. Changes inside the function **affect the original structure**.

Example:

```
#include <stdio.h>

struct Student {
int id;
char name[20];
float marks;
};

void changeMarks(struct Student *s, float newMarks) {
s->marks = newMarks; // Use -> for pointer to structure
}

int main() {
struct Student s1 = {1, "Bob", 80.5};

changeMarks(&s1, 95.0); // Pass address of s1

printf("Updated Marks: %.2f\n", s1.marks); // Now 95.0

return 0;
}
```

Output:

```
Updated Marks: 95.00
```

Returning a Structure from a Function

A function can **return a structure**, which allows you to **create a structure inside a function and send it back** to the caller.

Example:

```
#include <stdio.h>

struct Student {
int id;
char name[20];
float marks;
};
```

```
struct Student createStudent(int id, char name[], float marks) {
struct Student s;
s.id = id;
strcpy(s.name, name);
s.marks = marks;
return s;
}

int main() {
struct Student s1;

s1 = createStudent(2, "Alice", 90.0);

printf("ID: %d\n", s1.id);
printf("Name: %s\n", s1.name);
printf("Marks: %.2f\n", s1.marks);

return 0;
}
```

Output:

```
ID: 2
Name: Alice

Marks: 90.00
```

7.Explain functions with structured result values using struct.

Functions with Structured Result Values in C

In C, a **structure** (struct) is a user-defined data type that groups variables of different types under a single name. Functions in C can **return a structure**, which allows returning **multiple values** from a function in a clean and organized way.

Functions with Structured Result Values (using struct)

In **C language**, a function can **return a structure** as its result value. This is useful when a function needs to return **multiple related values** together.

Explanation

- A structure is defined using the struct keyword.
- The function’s return type is declared as that structure type.

- Inside the function, a structure variable is created, values are assigned, and the structure is returned.
- The calling function receives the returned structure and accesses its members.

Example: Returning a Structure from a Function

```
#include <stdio.h>

struct Result {

    int sum;

    int product;

};

struct Result calculate(int a, int b) {

    struct Result r;

    r.sum = a + b;

    r.product = a * b;

    return r;

}

int main() {

    struct Result res;

    res = calculate(5, 3);

    printf("Sum = %d\n", res.sum);

    printf("Product = %d\n", res.product);

    return 0;

}
```

OUTPUT :

```
Sum = 8
```

Product = 15

8.Give an example of a union and explain how memory is shared.

A **union** in C is similar to a structure (struct) but with one key difference:

- In a **struct**, all members have their own memory and can store values independently.
- In a **union**, **all members share the same memory location**, so **only one member can hold a value at a time**.

Use case: Unions are useful when you want a variable that can store **different types of data at different times**, saving memory.

Syntax of a Union

```
#include <stdio.h>

union Data {
int i;
float f;
char c;
};
```

- Here, Data is a union with **int, float, and char** members.
- All members share the **same memory location**.
- The size of the union is equal to the **largest member**.

Declaring a Union

A union is declared similarly to a structure:

```
union UnionName {
dataType1 member1;
dataType2 member2;
...
};
```

Accessing Union Members

```
#include <stdio.h>

// Define a union
union Student {
int age;
char grade;
};
```

```
int main() {
union Student s;

// Store age
s.age = 20;
printf("Student Age: %d\n", s.age);

// Store grade (overwrites age)
s.grade = 'A';
printf("Student Grade: %c\n", s.grade);

// Accessing age now will show garbage
printf("Student Age after storing grade: %d\n", s.age);

return 0;
}
```

Sample Output

```
Student Age: 20
Student Grade: A
Student Age after storing grade: 65   // Garbage value
```

Size of a Union

- The **size of a union** is equal to the **size of its largest member**.
- Smaller members share memory space without overflow.

Example:

```
#include <stdio.h>

union A {
int x;
char y;
};

union B {
int arr[10];
char y;
};

int main() {
printf("Sizeof A: %ld\n", sizeof(union A)); // 4
printf("Sizeof B: %ld\n", sizeof(union B)); // 40
```

return 0;
}

How Memory Is Shared in a Union

- All members of a union **use the same memory block**.
- The **size of the union** is equal to the size of its **largest data member**.

For the above union:

- int → 4 bytes
- float → 4 bytes
- char → 1 byte

So, the **union size = 4 bytes** (maximum of all members).

UNIT - V

1.Explain the difference between text and binary files in C.

A **file** is a collection of data stored in secondary memory, used for storing information that can be processed. Files allow data to be saved and retrieved later. They can store both programs and data, making file input and output operations crucial for reading from and writing to files.

There are two types of files in C language which are as follows

- Text file
- Binary File

Text File

- Text files contain alphabets and numbers that are easily understood by humans. Errors in text files can be easily identified and corrected. Each character in a text file is stored as one byte

C provides different functions to work with text files. Some functions are used to read the text files and some functions are used to write text file into a disk.

- Here are the functions

Function	Description
fscanf()	Reads input from the current file of the specified stream.
fgetc()	Reads a single character from the specified file, and increases the file position.
fgets()	This reads the single line of text or a string from the given file.
fprintf()	This prints the content in the file.
fputc()	This converts c to a different character and then outputs it at the current position.

fputs()	This converts a string of characters to a file at a specific location.
fwrite()	This allows us to write data to a binary file.

Binary files:

Binary files consist of 1's and 0's, which are easily understood by computers. Errors in binary files can also duplicate the file and these are very difficult to detect. In a binary file, the integer value 1245 determines 2 bytes both in the file and memory.

The steps to copy a binary file are as follows

- Open the source file in binary read mode.
- Open the destination file in binary write mode.
- Read a character from the file. The file pointer starts at the beginning, so no need to position it.
- If feof() indicates the end of the file, then close both files and return to the calling program.
- If the end of the file is not reached, then write the character to the destination file and repeat the process.

2.How do you open, read, and write a file in C?

A **file** in C is a collection of data stored permanently on a storage device (such as a hard disk).
Files are used to **store data permanently**, unlike variables which store data temporarily in memory.

C provides file handling through the FILE structure defined in the header file <stdio.h>.
File operations are performed using file pointers.

Steps to Open, Read, and Write a File in C

1) Opening a File

A file is opened using the fopen() function.

Syntax:

```
FILE *fp;

fp = fopen("filename", "mode");
```

Common File Modes:

Mode	Description
"r"	Open file for reading
"w"	Open file for writing (creates new file)
"a"	Open file for appending
"r+"	Read and write
"w+"	Write and read

2) Writing to a File

Data is written to a file using fprintf(), fputs(), or fputc().

Example:

```
fprintf(fp, "Hello World");
```

3) Reading from a File

Data is read from a file using fscanf(), fgets(), or fgetc().

Example:

```
fscanf(fp, "%s", text);
```

4) Closing a File

After completing operations, the file must be closed using fclose().

Syntax:

```
fclose(fp);
```

Example: Open, Write, and Read a File in C

```
#include <stdio.h>


int main() {
    FILE *fp;

    char text[50];

    // Writing to a file

    fp = fopen("data.txt", "w");

    fprintf(fp, "Welcome to C file handling");

    fclose(fp);

    // Reading from a file

    fp = fopen("data.txt", "r");

    fscanf(fp, "%[^\n]", text);

    printf("File Content: %s\n", text);

    fclose(fp);

    return 0;

}
```

Output

```
File Content: Welcome to C file handling
```

3.What are file pointers? Explain fopen, fclose, fread, and fwrite.

A **file pointer** is a pointer variable of type FILE * that refers to a file and is used to perform file operations such as reading, writing, and appending.

File pointers are declared using the FILE structure defined in the header file:

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

Declaration

```
FILE *fp;
```

Why File Pointers Are Needed

- To identify a file uniquely
- To keep track of:
  - File name
  - File access mode
  - Current position in the file
  - Buffer information
- To connect the program with the file stored on disk

Without a file pointer, a C program cannot access a file.

1. fopen() – Opening a File

fopen() opens a file and connects it with a file pointer.

Syntax

FILE *fopen(const char *filename, const char *mode);

Parameters

1. filename → Name of the file (with path if needed)
2. mode → Purpose for opening the file (read/write/append)

Return Value

- Returns a valid file pointer if successful
- Returns NULL if the file cannot be opened

File Opening Modes

Mode	Meaning
r	Open existing file for reading
w	Open file for writing (creates new or overwrites)
a	Open file for appending
r+	Read and write
w+	Write and read
a+	Read and append
rb	Read binary file

Mode	Meaning
wb	Write binary file
ab	Append binary file

2. fclose() – Closing a File

fclose() closes an opened file and releases system resources.

Syntax

int fclose(FILE *fp);

Return Value

- 0 → success
- EOF → failure

Why fclose() Is Important

- Saves data permanently
- Clears file buffers
- Prevents data loss
- Frees memory

Example

fclose(fp);

3. fread() – Reading Data from a File

fread() reads binary data from a file into memory.

Syntax

size_t fread(void *ptr, size_t size, size_t count, FILE *fp);

Parameters

1. ptr → Address where data is stored
2. size → Size of one data element (in bytes)

3. count → Number of elements to read
4. fp → File pointer

Return Value

- Number of elements successfully read

4. fwrite() – Writing Data to a File

fwrite() writes binary data from memory into a file.

Syntax

size_t fwrite(const void *ptr, size_t size, size_t count, FILE *fp);

Parameters

1. ptr → Address of data to be written
2. size → Size of each element
3. count → Number of elements
4. fp → File pointer

Return Value

- Number of elements successfully written

Program :

```
#include <stdio.h>

int main()

{

    FILE *fp;

    int num = 100, readNum;



    fp = fopen("number.bin", "wb");

    fwrite(&num, sizeof(int), 1, fp);
    fclose(fp);

    fp = fopen("number.bin", "rb");

    fread(&readNum, sizeof(int), 1, fp);

    fclose(fp);

    printf("Read number = %d", readNum);

    return 0;

}
```

Output :

```
Read number = 100
```

4.Write a program to read data from a text file and display it.

```
#include <stdio.h>

#include <stdlib.h>

int main() {

    FILE *fp;

    char ch;

    // Open the file in read mode

    fp = fopen("example.txt", "r");

    if (fp == NULL) {

        printf("Error: File not found!\n");

        return 1; // Exit program if file doesn't exist

    }

    printf("Contents of the file:\n");

    // Read and display characters until End of File (EOF)
    while ((ch = fgetc(fp)) != EOF) {

        putchar(ch);

    }

    // Close the file

    fclose(fp);

    return 0;

}
```

Output :

```
Hello, C Programming!

Welcome to file handling in C.
```

5.Explain linear search and implement it for an array of integers.

Linear search sequentially checks each element of the list until it finds a match for the keyvalue.
If the algorithm reaches the end of the list without finding the key, it reports that the element is not found.

Note:

Linear Search is also known as **Sequential Search**.

How it works

- Start from the first element.
- Compare each element with the key.
- If a match is found → return the position.
- If the end is reached → key not found.

Program

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

int main() {
int a[100], n, i, key;
printf("Enter how many elements you want to enter into the array: ");
scanf("%d", &n);

printf("Enter elements one by one into the array:\n");
for (i = 0; i < n; i++) {
scanf("%d", &a[i]);
}

printf("Enter key value: ");
scanf("%d", &key);

for (i = 0; i < n; i++) {
if (a[i] == key) {
printf("The element is found at position %d\n", i + 1);  // Position starts at 1
return 0;  // Exit program
}
}

printf("The element is not found.\n");
return 0;
}
```

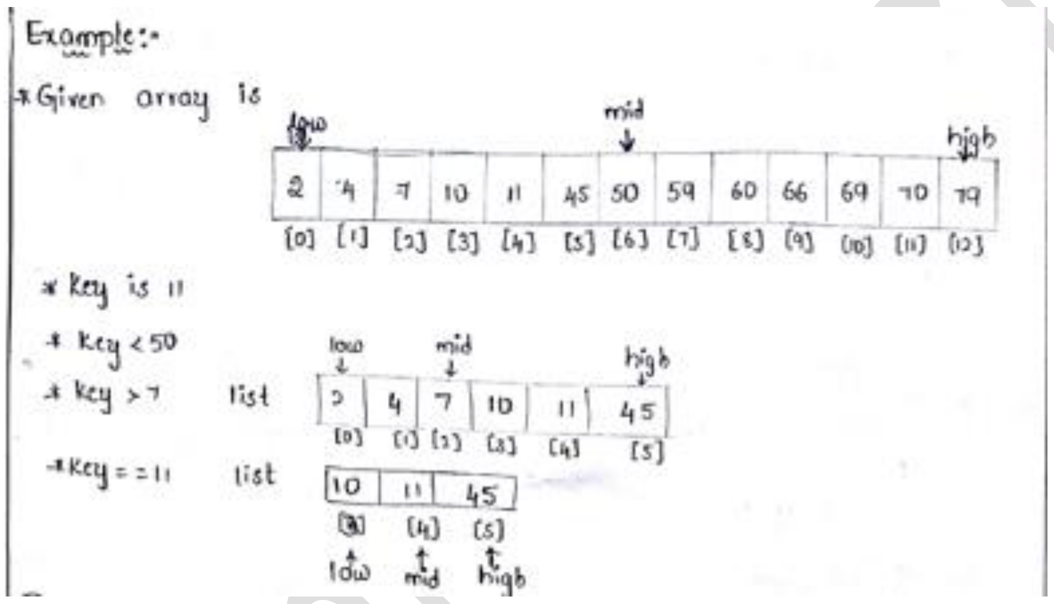
Output

```
Enter how many elements you want to enter into the array: 5

Enter elements one by one into the array:

10

25

30

45

50

Enter key value: 30

The element is found at position 3
```

6.Explain binary search and implement it for a sorted array.

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1.In this method divided into the given sorted list of elements will be 3 parts
2.by the first element treated as low
3.by the last element treated as high
4.And the mid =(low+high)/2
5.The key is compared with the middle value, if the element is found the key index is returned.
6.it doesn't match then the required element must be either in the left half (on) right half of the middle.
7. If the key is less than the middle value the key is searched in the left part else it is searched in the right part of the middle value.



Program:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int a[100], n, key, low, high, mid, i;
printf("Enter how many elements you want to enter into the array: ");
scanf("%d", &n);
printf("Enter elements one by one: ");
for (i = 0; i < n; i++) {
scanf("%d", &a[i]);
}
printf("Enter the key value: ");
scanf("%d", &key);
low = 0;
high = n - 1;
while (low <= high) {
mid = (low + high) / 2;
if (key == a[mid]) {
printf("The value is at position %d\n", mid + 1);
return 0;
}
else if (key < a[mid]) {
high = mid - 1;
}
else {
low = mid + 1;
}
}
printf("The element is not found\n");
return 0;
}
```

Sample Output:

```
Enter how many elements you want to enter into the array: 3
Enter elements one by one: 4
5
6
Enter the key value: 5
The value is at position 2
```

7.Explain bubble sort and write a program to sort an array.

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted. The largest (or smallest) elements "bubble up" to their correct position in each pass, hence the name "Bubble Sort."

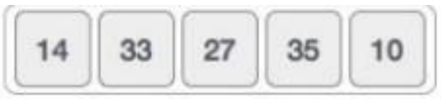
Algorithm

Following are the in bubble sort (for sorting a given array in ascending order).

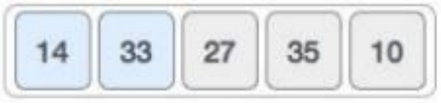
1.Starting with the first element (index=0), compare the current element with the next element of the array.

2.If the current element is greater than the next element of the array swap them.

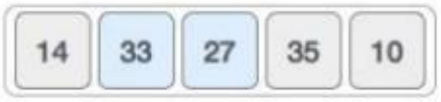
3.If the current element is less than the next element, of move to the next element. Repeat step 1.



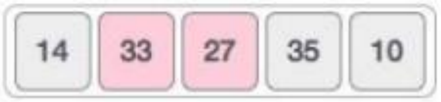
Bubble sort starts with very first two elements, comparing them to check which one is greater.



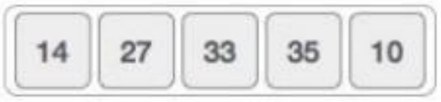
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



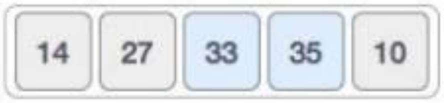
We find that 27 is smaller than 33 and these two values must be swapped.



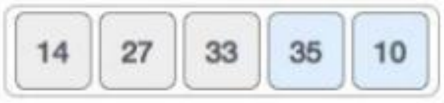
The new array should look like this −



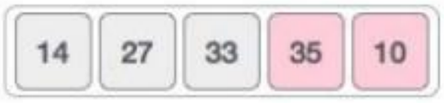
Next we compare 33 and 35. We find that both are in already sorted positions.



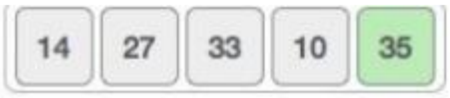
Then we move to the next two values, 35 and 10.



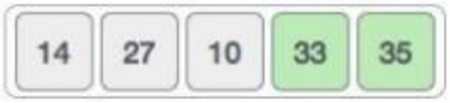
We know then that 10 is smaller 35. Hence they are not sorted.



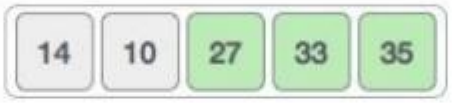
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



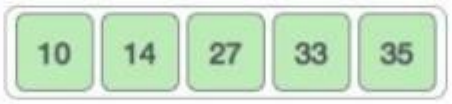
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Program:

```
#include <stdio.h>
void main() {
int a[100], n, i, j, temp;
printf("Enter how many elements you want to enter: ");
scanf("%d", &n);
printf("Enter elements one by one:\n");
for (i = 0; i < n; i++) {
scanf("%d", &a[i]);
}
for (i = 0; i < n - 1; i++) {
for (j = 0; j < n - 1 - i; j++) {
if (a[j] > a[j + 1]) {
temp = a[j];
a[j] = a[j + 1];
a[j + 1] = temp;
}
}  }
printf("The sorted elements using bubble sort are:\n");
for (i = 0; i < n; i++) {
printf("%d ", a[i]);
}
getch();
}
```

Sample output:

```
Enter how many elements you want to enter: 5
Enter elements one by one:
4
5
```

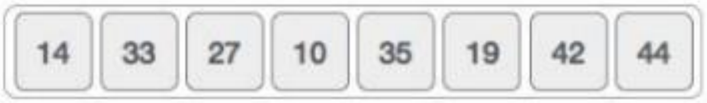
2
6
2
The sorted elements using bubble sort are:
                1   2 4 5 6

8.Explain insertion sort and give a simple program.

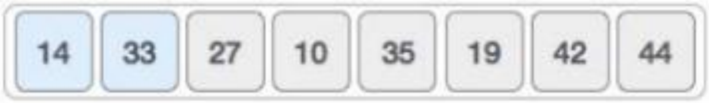
Insertion Sort is a simple and efficient sorting algorithm that builds the final sorted array one element at a time. It works by picking each element from the unsorted portion of the array and inserting it into its correct position within the sorted portion.
Algorithm:-
1.start
2. Declare the required variables.
3.the second element of the away is compared with the element that appears before it. if the second element is smaller than -first element, second element is inserted in the position of first element. After first step, first two elements are sorted.
4. The third element of an array is compared with the element that appear before it and If the third element is smaller -than second element, it is inserted in the position of ist element. if 3rd element is larger than first element but
5.smaller than second element, it is inserted in the position of second element. If 3rd element is larger then the position is as it is. This procedure is repeated until all elements are sorted in the array.

How Insertion Sort Works?

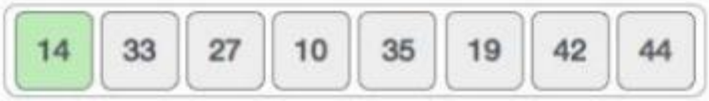
We take an unsorted array for our example.



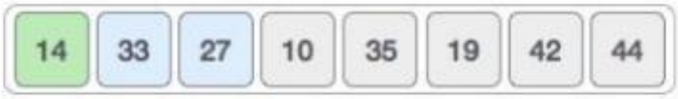
Insertion sort compares the first two elements.



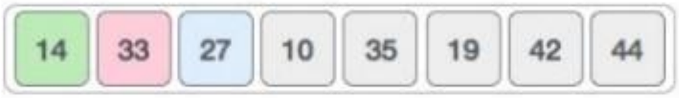
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



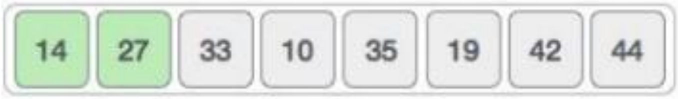
Insertion sort moves ahead and compares 33 with 27.



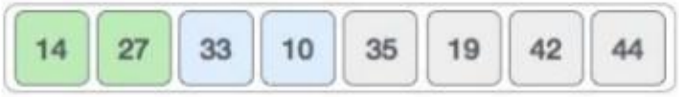
And finds that 33 is not in the correct position.



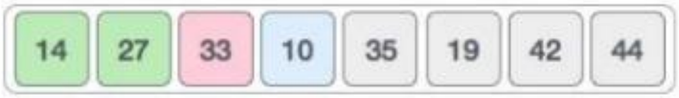
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



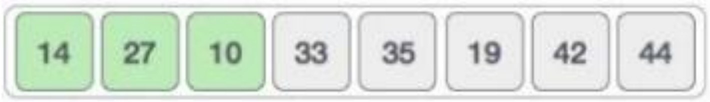
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



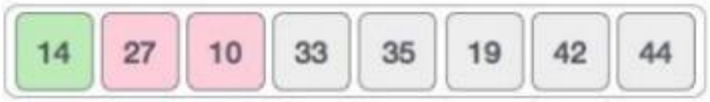
These values are not in a sorted order.



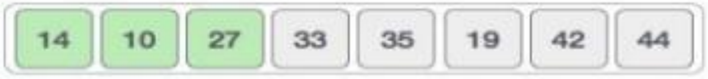
So we swap them.



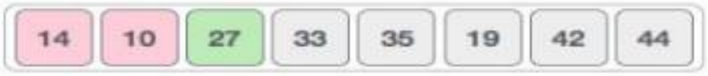
However, swapping makes 27 and 10 unsorted.



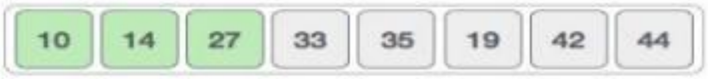
Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list

Program:

```
#include <stdio.h>
#include <conio.h>
void main() {
int a[100], n, i, key, temp;
printf("Enter how many elements you want to enter: ");
scanf("%d", &n);
printf("Enter the elements one by one:\n");
for (i = 0; i < n; i++) {
scanf("%d", &a[i]);
}
for (i = 1; i < n; i++) {
key = a[i];
int j = i - 1;
while (j >= 0 && a[j] > key) {
a[j + 1] = a[j];
j = j - 1;
}
a[j + 1] = key;
}
printf("The sorted elements using Insertion Sort are:\n");
for (i = 0; i < n; i++) {
printf("%d ", a[i]);
}
getch();
```

}

sample output:

```
Enter how many elements you want to enter: 3
Enter the elements one by one:
2
4
3
The sorted elements using Insertion Sort are:
2 3 4.
```

9.Explain selection sort with an example program.

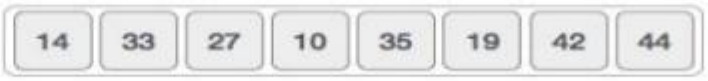
Selection Sort is a simple and intuitive sorting algorithm that sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and placing it in its correct position in the sorted portion.

Algorithm:

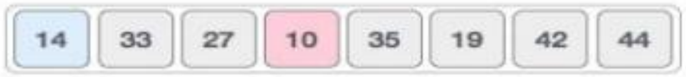
```
1.start
2.Declare the required variables.
3) Assume the first element is minimum, compare it with the next element. If it is greater swap them.     Every time the array is Compared with first element
 4) This is continued until no swaps required.
5) The elements are settled in the ascending order.
6)stop.
```

<u>How Selection Sort Works?</u>

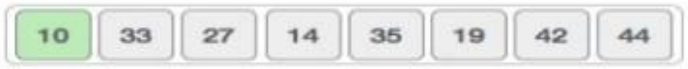
Consider the following depicted array as an example.



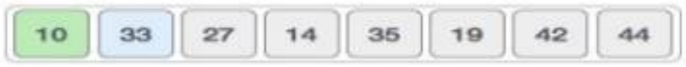
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



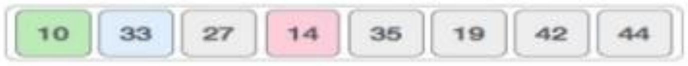
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



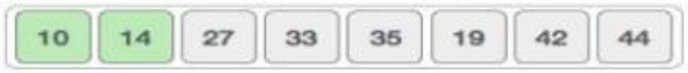
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

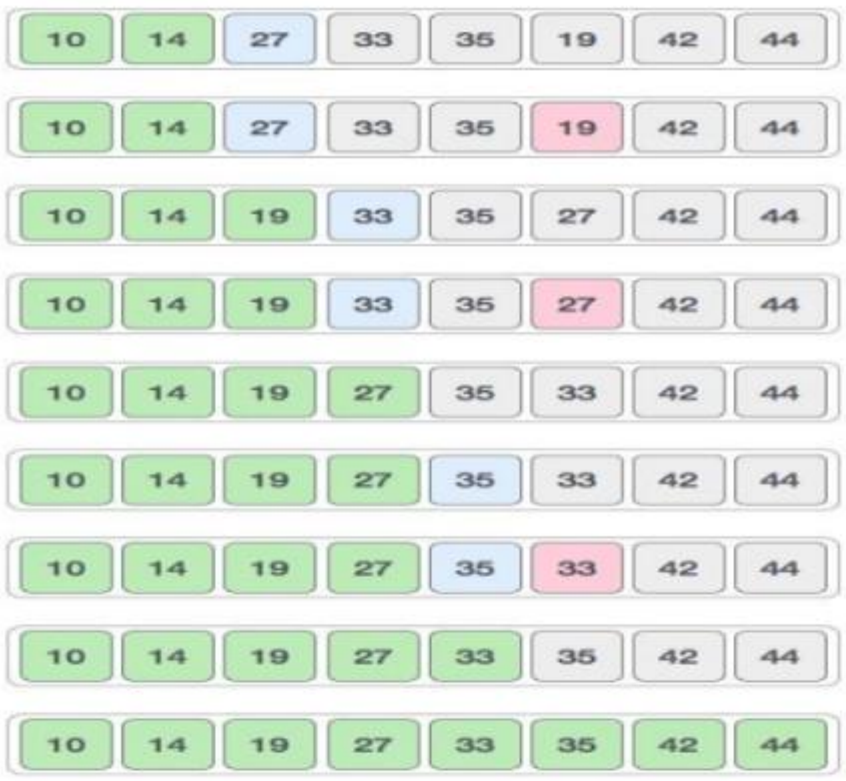


After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



Program:-

```
#include <stdio.h>
#include <conio.h>
void main() {
int a[100], n, i, j, temp, min;
printf("Enter how many elements you want to enter into array: ");
scanf("%d", &n);
printf("Enter the elements one by one:\n");
for (i = 0; i < n; i++) {
scanf("%d", &a[i]);
}
for (i = 0; i < n - 1; i++) {
min = i;

for (j = i + 1; j < n; j++) {
if (a[j] < a[min]) {
min = j;  // Update the minimum index if a smaller element is found
}
}
temp = a[i];
a[i] = a[min];
a[min] = temp;
}
printf("The sorted elements using selection sort are:\n");
for (i = 0; i < n; i++) {
printf("%d ", a[i]);
}
getch();
}
```

Sample output

```
Enter how many elements you want to enter into array: 4
Enter the elements one by one:
4
2
8
6
The sorted elements using selection sort are:
              2   4 6 8
```

10.What are the time complexities of insertion sort, bubble sort, and selection sort?

1.Selection Sort

Case	**Time Complexity**	**Explanation**
Best Case	$O(n^2)$	Always scans remaining elements to find minimum; comparisons remain the same even if array is already sorted.
Average Case	$O(n^2)$	Random order; comparisons and some swaps occur.
Worst Case	$O(n^2)$	Array in reverse order; maximum comparisons; swaps are minimal (1 per pass).

2. Insertion Sort

Case	**Time Complexity**	**Explanation**
Best Case	$O(n)$	Array already sorted; only one pass with comparisons, no shifting needed.
Average Case	$O(n^2)$	Random order; shifting required for some elements.
Worst Case	$O(n^2)$	Array in reverse order; maximum shifting required for each element.

3. Bubble Sort

Case	**Time Complexity**	**Explanation**
Best Case	$O(n)$	Optimized version: array already sorted; only one pass with no swaps.
Average Case	$O(n^2)$	Random order; comparisons and swaps occur.

Case	Time Complexity	Explanation
Worst Case	$O(n^2)$	Array in reverse order; maximum comparisons and swaps occur.

The end

All the best