1. Difference Between RAM and ROM

RAM (Random Access Memory) and ROM (Read-Only Memory) are both types of computer memory, but they serve different purposes. Here are the **major differences** between them:

Feature	RAM (Random Access Memory)	ROM (Read-Only Memory)
Definition	Volatile memory used for temporary data storage.	Non-volatile memory used for permanent storage.
Volatility	**Volatile** – Data is lost when power is turned off.	**Non-volatile** – Data remains even after power loss.
Usage	Stores data currently in use by the CPU (OS, applications).	Stores firmware (BIOS, bootloader, embedded systems).
Read/Write	**Read & Write** – Data can be modified frequently.	**Read-Only** – Data is written once (mostly during manufacturing).
Speed	**Faster** – Allows quick access for active processes.	**Slower** – Used for long-term storage, not for active processing.
Types	DRAM (Dynamic RAM), SRAM (Static RAM).	PROM, EPROM, EEPROM, Flash Memory.
Modification	Data can be changed anytime.	Data is permanent or requires special methods to modify (e.g., flashing).
Cost	More expensive per GB.	Cheaper for permanent storage.
Capacity	Higher capacity (up to several GBs in modern systems).	Smaller capacity (MBs to a few GBs).

Key Takeaways:

- **RAM** is **temporary, fast, and volatile** (used for active tasks).
- **ROM** is **permanent, slower, and non-volatile** (used for firmware).
- **RAM** loses data when powered off, while **ROM** retains it.
- **RAM** is used for running applications, whereas **ROM** stores critical system instructions.

2. Components of a Computer System & Their Roles in Program Execution

A computer system consists of **hardware** and **software** components that work together to execute programs. The major components and their roles in program execution are:

1. Input Unit

Function: Accepts data and instructions from the user or external devices.
Examples: Keyboard, Mouse, Scanner, Microphone.
Role in Program Execution:

- Provides the initial data/commands required for processing.
- Converts human-readable input into machine-readable form.

2. Central Processing Unit (CPU) – "Brain of the Computer"

Function: Performs arithmetic, logical, and control operations.
Subcomponents:

a) Control Unit (CU)

- Manages and coordinates all operations.
- Fetches instructions from memory, decodes them, and executes them.

b) Arithmetic Logic Unit (ALU)

- Performs arithmetic (+, -, *, /) and logical (AND, OR, NOT) operations.

c) Registers (Temporary Storage in CPU)

- Small, fast storage locations (e.g., **Program Counter (PC), Accumulator, Instruction Register**).
  Role in Program Execution:
- Executes instructions fetched from memory.
- Performs calculations and decision-making.

3. Memory Unit (Primary Storage)

a) RAM (Random Access Memory)

- **Volatile** (loses data when power is off).
- Stores **currently running programs and data**.
- Faster than secondary storage.

b) ROM (Read-Only Memory)

- **Non-volatile** (retains data permanently).

- Stores **firmware (BIOS, bootloader)**.
  Role in Program Execution:
- Holds **program instructions and data** while the CPU processes them.
- RAM allows quick access to active data, while ROM ensures the system boots correctly.

4. Secondary Storage (Auxiliary Memory)

Function: Stores data permanently even when the computer is turned off.
Examples: HDD, SSD, USB Drives, CDs.
Role in Program Execution:

- Stores the **operating system, software, and user files**.
- Loads programs into **RAM** when needed for execution.

5. Output Unit

Function: Displays or transmits processed data to the user.
Examples: Monitor, Printer, Speakers.
Role in Program Execution:

- Shows the **results** of computations (e.g., display, printout, sound).

6. System Bus (Data Transfer Pathway)

Function: Connects CPU, Memory, and I/O devices for communication.
Types:

- **Data Bus** (Transfers data between components).
- **Address Bus** (Carries memory addresses).
- **Control Bus** (Sends control signals like read/write).
  Role in Program Execution:
- Ensures smooth transfer of **instructions and data** between CPU, memory, and I/O devices.

How These Components Work Together in Program Execution?

1. **Input Unit** → Provides data/instructions.
2. **Memory (RAM/ROM)** → Stores the program and data.
3. **CPU (CU + ALU + Registers)** →
   - **Fetches** instructions from RAM.
   - **Decodes** the instructions.
   - **Executes** them (calculations in ALU).
4. **Secondary Storage** → Keeps programs/files for long-term use.

5. **Output Unit** → Displays the final result.

Summary (Key Points for Exams)

Component	Role in Program Execution
Input Unit	Takes user input.
CPU (CU, ALU, Registers)	Fetches, decodes, executes instructions.
RAM	Temporarily holds running programs/data.
ROM	Stores boot instructions (BIOS).
Secondary Storage	Stores OS, software, and files.
Output Unit	Displays results.
System Bus	Connects components for data transfer.

3. C Language: Definition and Characteristics

What is C Language?

C is a **procedural, high-level programming language** developed by **Dennis Ritchie** in 1972 at **Bell Labs**. It was designed for **system programming**, particularly for writing operating systems like **UNIX**.

C is often called the **"mother of all programming languages"** because many modern languages (C++, Java, Python) are influenced by its syntax and structure.

Key Characteristics of C Language

1. Procedural Programming

- Follows a **step-by-step approach** (functions and procedures).
- Programs are divided into **functions**, making them modular and easy to debug.

2. Mid-Level Language

- Combines features of **low-level (assembly)** and **high-level (Python, Java)** languages.
- Allows **bit manipulation** (like low-level) but also provides **structured programming** (like high-level).

3. Portability (Platform-Independent)

- C programs can run on different machines with little or no modification.
- Example: A program written on **Windows** can be compiled and run on **Linux**.

4. Fast and Efficient

- Provides **direct memory access** (pointers).
- Used in **system programming, embedded systems, and game development** due to its speed.

5. Rich Library Support

- Comes with a **standard library (stdio.h, math.h, string.h, etc.)** for common operations.
- Reduces coding effort (e.g., printf(), scanf()).

6. Static Typing

- Variables must be **declared with a data type** before use (e.g., int x;).
- Helps in **early error detection**.

7. Pointers for Memory Management

- Allows **direct memory manipulation** using pointers (int *p;).
- Useful for **dynamic memory allocation** (malloc(), free()).

8. Structured Language

- Supports **loops (for, while), conditionals (if-else), and functions** for better code organization.

9. Case-Sensitive

- int x; and Int X; are treated as different variables.

10. No Built-in OOP (Unlike C++, Java)

- Does not support **classes, objects, or inheritance** natively (but can be simulated using structures and functions).

4. Identifier in C .

An **identifier** is a name given to **variables, functions, arrays, structures**, etc., in a C program.

Rules for Declaring Identifier Names

1. **Valid Characters**:
    - Can include **letters (A-Z, a-z), digits (0-9), and underscore (_)**.
    - Must **start with a letter or underscore** (not a digit).
2. **Case-Sensitive**:
    - sum, Sum, and SUM are different identifiers.
3. **No Keywords**:
    - Cannot use **reserved words** like int, if, while, etc.

4. **Length Limit**:
    - Up to **31 characters** (compiler-dependent).
5. **No Special Symbols**:
    - Cannot use !, @, #, $, etc.

Examples

Valid: total, _count, sum1
Invalid: 5sum (starts with digit), float (keyword), first-name (hyphen not allowed)

5. Number System Conversions

a) $(101110)_2$ to Decimal

Process: Write each bit with its positional power of 2:

1×25+0×24+1×23+1×22+1×21+0×201×25+0×24+1×23+1×22+1×21+0×20

Calculate each term:

32+0+8+4+2+0=4632+0+8+4+2+0=46

Answer: 46104610

---

b) $(47)_8$ to Binary

Process: Octal → Binary (each octal digit = **3 binary bits**):

4 (Octal)	**7 (Octal)**
100	111

Combine the bits: 100 111100 111

Answer:10011121001112

---

c) $(2F)_{16}$ to Decimal

Process:

Convert each hex digit to decimal:

- 2 = 2
- F = 15

Multiply by powers of 16: 2×161+15×160=32+15=472×161+15×160=32+15=47

Answer: 47104710

---

d) $(1001)_2$ to Octal

Process:

Binary → Octal (group into **3 bits from right**): 1 0011 001

(Add leading zeros if needed: 001 001) Convert each group to octal:

001 (Binary)	**001 (Binary)**
1 (Octal)	1 (Octal)

1. Combine: 118118

Answer: 118118

---

e) $(59)_{10}$ to Binary

Process:

Divide by 2 and record remainders:

59÷2=29Remainder 129÷2=14Remainder 114÷2=7Remainder 07÷2=3Remainder 13÷2=1Remainder 1 1÷2=0Remainder 159÷2=2929÷2=1414÷2=77÷2=33÷2=11÷2=0
Remainder 1Remainder 1Remainder 0Remainder 1Remainder 1Remainder 1

Read remainders from bottom to top:11101121110112

Answer: 11101121110112

6. Type Casting in C Programming

Type casting is the process of converting a variable from one data type to another. It allows programmers to **temporarily change the type of a variable** for a specific operation.

Types of Type Casting

1. Implicit Type Casting (Automatic Conversion)

- Done **automatically by the compiler** when a lower data type is assigned to a higher data type.
- Follows the **data type hierarchy**:

char → int → long → float → doublechar → int → long → float → double

- **Example:**

```
int a = 5;
```

```
float b = a;  // int is implicitly converted to float
```

2. Explicit Type Casting (Manual Conversion)

- Done **manually by the programmer** using the **cast operator** (datatype).
- Used when **loss of data** might occur (e.g., converting float to int).
- **Example:**

```
float x = 5.7;

int y = (int)x;  // Explicitly converts float to int (y = 5)
```

Why Use Type Casting?

✓ **Precision Control** (e.g., forcing integer division to return a float).
✓ **Avoid Data Loss** (e.g., converting float to int carefully).
✓ **Function Compatibility** (e.g., passing int to a function expecting float).

Examples

Example 1: Implicit Casting

```
int num1 = 10;

float num2 = 5.5;

float result = num1 + num2;  // num1 is implicitly converted to float

printf("%f", result);  // Output: 15.5
```

Example 2: Explicit Casting

```
float a = 9.8;

int b = (int)a;  // Explicitly converts float to int (b = 9)

printf("%d", b);  // Output: 9
```

Example 3: Avoiding Integer Division

```
int x = 5, y = 2;

float result = (float)x / y;  // Explicit cast to get float division

printf("%f", result);  // Output: 2.5 (instead of 2)
```

Key Points to Remember

- **Implicit casting** is safe (no data loss).
- **Explicit casting** may lead to **data loss** (e.g., float → int truncates decimal part).

- **Casting does not modify the original variable**, it only creates a temporary converted value.

Summary Table

Type Casting	How It Works	Example
Implicit	Automatic conversion by compiler	int → float
Explicit	Manual conversion using (type)	(int) 5.7 → 5

7. Difference Between Algorithm and Flowchart

(Common 1st-year B.Tech Question in Programming Fundamentals)

Feature	Algorithm	Flowchart
Definition	Step-by-step procedure to solve a problem.	Graphical representation of an algorithm using symbols.
Form	Textual (written in simple language or pseudocode).	Visual (uses shapes like rectangles, diamonds, arrows).
Purpose	Focuses on **logic** and **process**.	Focuses on **visual clarity** and **flow of steps**.
Ease of Understanding	Requires basic programming knowledge.	Easier to understand (even for non-programmers).
Modification	Easier to modify (text-based).	Harder to modify (requires redrawing).
Use Case	Used in **planning and coding**.	Used in **documentation and presentations**.
Examples	**Pseudocode** for finding the largest number.	**Diagram** with start/end, input/output, decision boxes.

Key Takeaways

✓ **Algorithm** = **Logical steps** (text-based).
✓ **Flowchart** = **Visual diagram** (symbol-based).
✓ **Algorithms** are better for **coding**, while **flowcharts** are better for **explaining** the process.

8. Types of Operators in C (with Examples)

Operators in C are symbols that perform operations on variables and values. They are classified into **7 main types**:

1. Arithmetic Operators

Used for **mathematical calculations**.

Operator	**Description**	**Example (**int a=5, b=2**)**	**Result**
+	Addition	a + b	7
-	Subtraction	a - b	3
*	Multiplication	a * b	10
/	Division	a / b	2 (integer division)
%	Modulus (remainder)	a % b	1

2. Relational Operators

Compare two values (**returns** 1 **(true) or** 0 **(false)**).

Operator	**Description**	**Example (**a=5, b=2**)**	**Result**
==	Equal to	a == b	0
!=	Not equal	a != b	1
>	Greater than	a > b	1
<	Less than	a < b	0
>=	Greater than or equal	a >= b	1
<=	Less than or equal	a <= b	0

3. Logical Operators

Combine multiple conditions (**returns** 1 **(true) or** 0 **(false)**).

Operator	**Description**	**Example (**a=1**,** b=0**)**	**Result**
&&	Logical AND	a && b	0
||	Logical OR	a || b	1
!	Logical NOT	!a	0

4. Assignment Operators

Assign values to variables.

Operator	Description	Example	Equivalent to
=	Simple assignment	a = 5	a = 5
+=	Add and assign	a += 3	a = a + 3
-=	Subtract and assign	a -= 2	a = a - 2
*=	Multiply and assign	a *= 4	a = a * 4
/=	Divide and assign	a /= 2	a = a / 2

5. Increment/Decrement Operators

Change the value of a variable by 1.

Operator	Description	Example (int a=5)	Result
++	Increment	a++ or ++a	a = 6
--	Decrement	a-- or --a	a = 4

- **Prefix (++a)**: Increment first, then use.
- **Postfix (a++)**: Use first, then increment.

6. Bitwise Operators

Perform operations on **binary bits**.

Operator	Description	Example (a=5, b=3)	Result (Binary)
&	Bitwise AND	a & b	101 & 011 = 001 (1)
|	Bitwise OR	a | b	101 | 011 = 111 (7)
^	Bitwise XOR	a ^ b	101 ^ 011 = 110 (6)
~	Bitwise NOT	~a	~101 = 010 (2)
<<	Left shift	a << 1	101 → 1010 (10)
>>	Right shift	a >> 1	101 → 010 (2)

7. Conditional (Ternary) Operator

Shortcut for if-else.
Syntax: condition ? expression1 : expression2

Example:

```
int a = 5, b = 2;

int max = (a > b) ? a : b;  // max = 5
```

Special Operators

- sizeof(): Returns size of a variable.

```
printf("%d", sizeof(int));  // Output: 4 (bytes)
```

- **Comma Operator (,)**: Separates expressions.

```
int a = (2, 3, 4);  // a = 4 (last value)
```

9. Number System Conversions

1. Binary to Decimal Conversion

Process:
Multiply each bit by 2^n (where n*n* is its position from **right, starting at 0**) and sum the results.

Example: Convert (1011)2(1011)2 to decimal.

$$1 \times 2^3 = 8$$
$$0 \times 2^2 = 0$$
$$1 \times 2^1 = 2$$
$$1 \times 2^0 = 1$$
$$\text{Total} = 8 + 0 + 2 + 1 = \boxed{11_{10}}$$

2. Binary to Octal Conversion

Process:

1. Group binary digits into sets of **3** (from right).
2. Convert each group to its octal equivalent.

Example: Convert $(110101)_2$(110101)2 to octal.

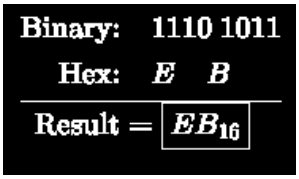
Binary:	110 101
Octal:	6 5
Result =	$\boxed{65_8}$

3. Binary to Hexadecimal Conversion

Process:

1. Group binary digits into sets of **4** (from right).
2. Convert each group to its hex equivalent (A=10, B=11, ..., F=15).

Example: Convert (11101011)2(11101011)2 to hexadecimal.



Key Conversion Table

Binary	Octal (3-bit)	Hexadecimal (4-bit)
000	0	0
001	1	1
010	2	2
011	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	-	8
1001	-	9
1010	-	A
1011	-	B
1100	-	C
1101	-	D
1110	-	E
1111	-	F

Why Learn These Conversions?

- **Binary → Decimal**: Used in arithmetic operations.
- **Binary → Octal/Hex**: Simplifies representation of large binary numbers (e.g., memory addresses).
- Essential for **microprocessor programming** and **digital circuits**.

10. Operator Precedence and Associativity in C

1. Operator Precedence

Definition: The order in which operators are evaluated in an expression. Higher precedence operators are evaluated first.

Precedence Table (Highest to Lowest)

Category	Operators	Associativity
Parentheses	()	Left-to-Right
Unary	++, --, !, ~, +, - (unary)	Right-to-Left
Multiplicative	*, /, %	Left-to-Right
Additive	+, -	Left-to-Right
Shift	<<, >>	Left-to-Right
Relational	<, <=, >, >=	Left-to-Right
Equality	==, !=	Left-to-Right
Bitwise AND	&	Left-to-Right
Bitwise XOR	^	Left-to-Right
Bitwise OR	|	Left-to-Right
Logical AND	&&	Left-to-Right
Logical OR	||	Left-to-Right
Conditional	?:	Right-to-Left
Assignment	=, +=, -=, *=, /= etc.	Right-to-Left
Comma	,	Left-to-Right

2. Associativity

Definition: The direction (left-to-right or right-to-left) in which operators of the **same precedence** are evaluated.

- **Left-to-Right (L-R):** Most operators (e.g., +, -, *, /).
- **Right-to-Left (R-L):** Unary, assignment, conditional operators.

3. Detailed Example

Expression:

```
int result = 5 + 3 * 2 < 10 && 4 % 3 == 1;
```

Step-by-Step Evaluation:

1. **Parentheses:** None here.
2. **Unary Operators:** None here.
3. **Multiplicative (*, /, %):**
   - 3 * 2 = 6
   - 4 % 3 = 1
     Now: 5 + 6 < 10 && 1 == 1
4. **Additive (+, -):**
   - 5 + 6 = 11
     Now: 11 < 10 && 1 == 1
5. **Relational (<, <=, >, >=):**
   - 11 < 10 → 0 (false)
     Now: 0 && 1 == 1
6. **Equality (==, !=):**
   - 1 == 1 → 1 (true)
     Now: 0 && 1
7. **Logical AND (&&):**
   - 0 && 1 → 0 (false)
8. **Assignment (=):**
   - result = 0

Final Answer:

```
result = 0; // The expression evaluates to false (0).
```

---

4. Key Takeaways

✓ **Precedence** decides which operator is evaluated **first**.
✓ **Associativity** decides the order when operators have the **same precedence**.
✓ **Parentheses** () can override precedence.
✓ **Right-to-left associativity** is used for assignments (=, +=) and unary operators (++, --).

Common Pitfalls

1. **Incorrect assumption about precedence:**

```
int x = 5 * 2 + 3; // 13 (NOT 25, because * has higher precedence than +)
```

2. **Ignoring associativity:**

```
int y = 10;

int z = y = 5; // z = 5 (right-to-left associativity for =)
```

11. Write an algorithm and draw a flowchart to check whether a given number is prime or not.

Algorithm to Check Prime Number

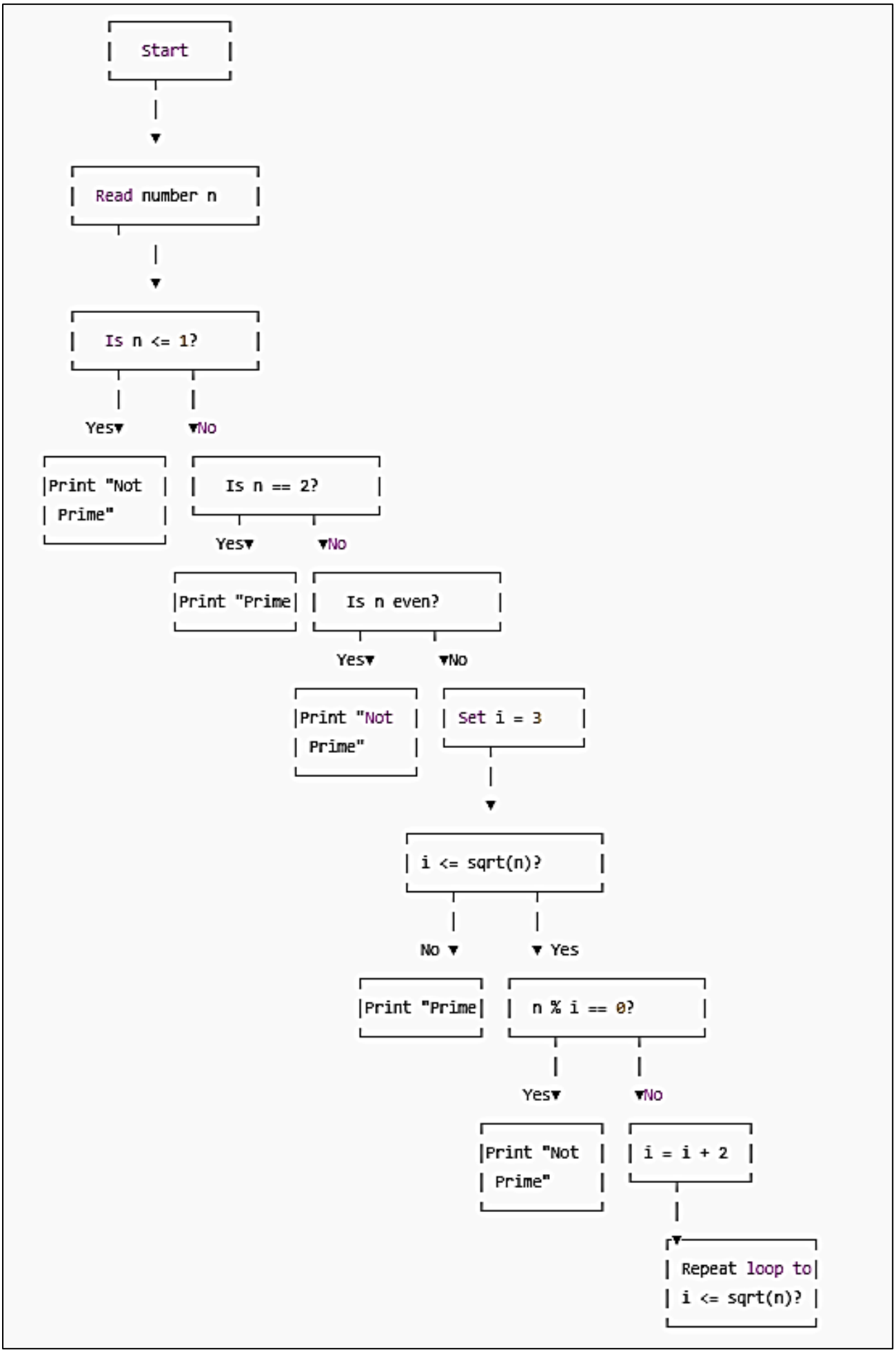
Input: A number n
Output: "Prime" or "Not Prime"

Steps:

1. Start
2. Read the number n
3. If n <= 1, print "Not Prime" and exit
4. If n == 2, print "Prime" and exit
5. If n is even, print "Not Prime" and exit
6. Set iWhile i <= sqrt(n):
    - If n % i == 0, print "Not Prime" and exit
    - Increment i by 2 (check only odd divisors)
7. If no divisors found, print "Prime"

8. Stop

Flowchart for Prime Numbe Check ----------→


Start
Read number n
Is n <= 1?
Yes
No
Print "Not Prime"
Is n == 2?
Yes
No
Print "Prime"
Is n even?
Yes
No
Print "Not Prime"
Set i = 3
i <= sqrt(n)?
No
Yes
Print "Prime"
n % i == 0?
Yes
No
Print "Not Prime"
i = i + 2
Repeat loop to i <= sqrt(n)?

12. C Program to Find Maximum Between Two Numbers Using Conditional (Ternary) Operator

Approach

1. **Input:** Take two numbers as input.
2. **Condition Check:** Use the ternary operator (condition) ? expression1 : expression2 to compare the numbers.
3. **Output:** Print the maximum number.

Program Code

```c
#include <stdio.h>

int main() {

  int num1, num2, max;

  // Input two numbers

  printf("Enter two numbers: ");

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

  // Find maximum using ternary operator

  max = (num1 > num2) ? num1 : num2;

  // Print the result

  printf("Maximum between %d and %d is: %d\n", num1, num2, max);

  return 0;

}
```

Explanation

1. **Input:**
    - scanf() reads two integers (num1 and num2).
2. **Ternary Operator Logic:**

- (num1 > num2) ? num1 : num2 checks if num1 is greater than num2.
    - If **true**, it assigns num1 to max.
    - If **false**, it assigns num2 to max.

3. **Output:**
    - printf() displays the maximum number.

Sample Output

Enter two numbers: 25 40

Maximum between 25 and 40 is: 40

Key Points

✓ **Ternary Operator** is a shorthand for if-else.
✓ Efficient for simple conditional checks.
✓ Works for all data types (int, float, etc.).

Alternative (Without Extra Variable)

```
printf("Maximum is: %d\n", (num1 > num2) ? num1 : num2);
```

13. Differentiate between the for, while, and do-while loops with suitable examples

Loop	Initialization	Condition Check	Minimum Runs	Typical Use Case
for	Inside ()	Before iteration	0	Counting (e.g., for(i=0;i<5;i++))
while	Outside	Before iteration	0	Input validation (e.g., while(x>0))
do-while	Outside	After iteration	1	Menus (e.g., do {...} while(choice!=0);)

Key Differences

1. **Initialization:**
    - for: Done inside the loop.
    - while/do-while: Done outside.
2. **Condition Check:**
    - for/while: Check **before** execution (may run **0 times**).
    - do-while: Checks **after** execution (runs **at least once**).
3. **Termination:**
    - All loops terminate when the condition becomes **false**.

Practical Examples

1. for **Loop (Fixed Iterations)**

```
// Print numbers 1 to 10
for(int i=1; i<=10; i++) {
  printf("%d ", i);
}
```

Output:

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

2. while **Loop (Condition-Based)**

```
// User input until a negative number is entered
int num;
printf("Enter a number: ");
scanf("%d", &num);
while(num >= 0) {
  printf("You entered: %d\n", num);
  printf("Enter another number: ");
  scanf("%d", &num);
```

```
}
```

Output:

```
Enter a number: 5

You entered: 5

Enter another number: -1
```

(Exits loop)

3. do-while **Loop (Guaranteed Execution)**

```
// Menu-driven program (runs at least once)

int choice;

do {

  printf("1. Start\n2. Exit\nEnter choice: ");

  scanf("%d", &choice);

 } while(choice != 2);
```

Output:

```
1. Start

2. Exit

Enter choice: 1

1. Start

2. Exit

Enter choice: 2
```

(Exits loop)

When to Use Which Loop?

- for**:** Counting, arrays, fixed iterations (e.g., for(i=0; i<n; i++)).

- while**:** Unknown iterations (e.g., reading input until valid).
- do-while**:** Menus, input validation (e.g., "Enter password").

14. Write a C program to check whether a number is prime or not using a for loop.

C program to check if a number is prime using a for **loop**, with clear explanations:

```c
#include <stdio.h>
#include <math.h> // For sqrt() function
int main() {
  int n, i, isPrime = 1; // isPrime=1 means true (assume prime initially)
  // Input
  printf("Enter a positive integer: ");
  scanf("%d", &n);
  // Edge cases
  if (n <= 1) {
    isPrime = 0; // 0 and 1 are not prime
  }
  else {
    // Check divisibility from 2 to sqrt(n)
    for (i = 2; i <= sqrt(n); i++) {
      if (n % i == 0) {
        isPrime = 0; // Found a divisor, not prime
        break;
      }
    }
  }
  // Output
```

```
  if (isPrime)
    printf("%d is a prime number.\n", n);
  else
    printf("%d is not a prime number.\n", n);
  return 0;
}
```

Key Features:

1. **Efficiency**: Only checks divisors up to $\sqrt{n}$ (mathematically optimal)
2. **Edge Handling**: Explicitly checks for $n \leq 1$
3. **Early Exit**: Uses break when a divisor is found
4. **Readable Logic**: Uses a isPrime flag for clear true/false tracking

Sample Outputs:

```
Enter a positive integer: 17
17 is a prime number.
Enter a positive integer: 15
15 is not a prime number.
```

15. Explain the concept of nested loops in C with an example.

Nested Loops in C

A **nested loop** is a loop inside another loop. The **inner loop** completes all its iterations for each iteration of the **outer loop**.
This is useful for **multi-dimensional patterns, matrix operations, and grid-based problems**.

Syntax

```
for (initialization; condition; update) {  // Outer Loop
  for (initialization; condition; update) { // Inner Loop
```

```
    // Code to execute
  }
}
```

(Works similarly with while and do-while loops.)

Example: Print a 3x3 Number Grid

```
#include <stdio.h>

int main() {
  int rows = 3, cols = 3;


  for (int i = 1; i <= rows; i++) {     // Outer loop (rows)
    for (int j = 1; j <= cols; j++) {  // Inner loop (columns)
      printf("%d ", i * j);        // Print product of row & column
    }
    printf("\n");  // New line after each row
  }
  return 0;
}
```

Output:

```
1 2 3
2 4 6
3 6 9
```

How It Works?

1. **Outer Loop (Rows):** Runs 3 times (i = 1, 2, 3).
2. **Inner Loop (Columns):** For each i, runs 3 times (j = 1, 2, 3).
    - Prints i * j (e.g., when i=2, prints 2*1=2, 2*2=4, 2*3=6).
3. **Newline:** After each row, printf("\n") moves to the next line.

Common Use Cases

1. **Pattern Printing** (e.g., stars, numbers).

```
// Print 5x5 star pattern
for (int i = 0; i < 5; i++) {
   for (int j = 0; j < 5; j++) {
      printf("* ");
   }
   printf("\n");
}
```

Output:

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

2. **Matrix Operations** (e.g., addition, multiplication).

```
// Add two 2x2 matrices
for (int i = 0; i < 2; i++) {
   for (int j = 0; j < 2; j++) {
      C[i][j] = A[i][j] + B[i][j];
```

```
  }
}
```

3. **Multi-dimensional Data Processing** (e.g., tables, grids).

Key Points

- **Inner loop completes fully** for each outer loop iteration.
- **Total iterations** = (Outer loop iterations) × (Inner loop iterations).
- **Break/Continue**: Affects only the innermost loop unless specified.

Practice Problem

Print a right-angled triangle of numbers:

```
1
1 2
1 2 3
1 2 3 4
```

Solution:

```
for (int i = 1; i <= 4; i++) {
  for (int j = 1; j <= i; j++) {
    printf("%d ", j);
  }
  printf("\n");}
```

16 Explain the use of break and continue statements in loops with examples..

break Statement

The break statement is used to *terminate* the loop immediately when it is encountered. Once break is executed, the loop stops, and the program control flows to the statement immediately following the loop.

Imagine you're searching for something specific in a list. Once you find it, there's no need to continue looking through the rest of the list, right? That's where break comes in handy.

Syntax:

```
# Inside a loop (for or while)

if condition:

  break
```

Example 1: Using break in a for loop (Searching for a number)

Suppose you have a list of numbers and you want to stop processing once you find the number 5.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print("--- Using break in a for loop ---")

for num in numbers:

  if num == 5:

    print(f"Found {num}! Breaking out of the loop.")

    break  # Exit the loop immediately

  print(f"Current number: {num}")



print("Loop finished.")
```

Output:

```
Current number: 1

Current number: 2

Current number: 3

Current number: 4

Found 5! Breaking out of the loop.

Loop finished.
```

Explanation: The loop iterates through numbers. When num becomes 5, the if condition is true, break is executed, and the loop terminates. The numbers 6, 7, 8, 9, 10 are not processed.

Example 2: Using break in a while loop (User input until 'quit')

```
print("\n--- Using break in a while loop ---")

while True: # An infinite loop

  user_input = input("Enter something (type 'quit' to exit): ")

  if user_input.lower() == 'quit':

    print("You typed 'quit'. Exiting the loop.")

    break # Exit the loop

  print(f"You entered: {user_input}")



print("Program continues after the while loop.")
```

Explanation: This while True loop would run forever if not for break. When the user types "quit" (case-insensitive), the break statement is executed, and the loop terminates.

continue Statement

The continue statement is used to *skip* the rest of the code inside the current iteration of the loop and move to the next iteration. It does not terminate the loop entirely; it just skips the current "cycle" of the loop.

Imagine you're processing a list of items, but some items need to be skipped due to certain conditions. You don't want to stop the entire process, just bypass the problematic item for that one round.

Syntax:

```
# Inside a loop (for or while)

if condition:

  continue
```

Example 1: Using continue in a for loop (Skipping even numbers)

Suppose you want to print only the odd numbers from a list.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]



print("\n--- Using continue in a for loop ---")

for num in numbers:
```

```
  if num % 2 == 0: # Check if the number is even
     print(f"Skipping even number: {num}")
     continue # Skip the rest of the current iteration and move to the next num
  printf("Processing odd number: {num}")


print("Loop finished processing numbers.")
```

Output:

```
Processing odd number: 1
Skipping even number: 2
Processing odd number: 3
Skipping even number: 4
Processing odd number: 5
Skipping even number: 6
Processing odd number: 7
Skipping even number: 8
Processing odd number: 9
Skipping even number: 10
Loop finished processing numbers.
```

Explanation: When num is an even number, the if condition is true, and continue is executed. This causes the print(f"Processing odd number: {num}") statement to be skipped for that particular even number, and the loop immediately proceeds to the next num.

Example 2: Using continue in a while loop (Skipping specific input)

```
printf("\n--- Using continue in a while loop ---")
count = 0
while count < 5:
  count += 1
```

```
  if count == 3:
     printf(f"Skipping iteration when count is {count}")
     continue # Skip the rest of this iteration
  printf(Current count: {count}")
printf("While loop finished.")
```

Output:

```
Current count: 1
Current count: 2
Skipping iteration when count is 3
Current count: 4
Current count: 5
While loop finished.
```

Explanation: When count becomes 3, the continue statement is executed. This means print(f"Current count: {count}") is skipped for count = 3, and the loop immediately moves to the next iteration (where count becomes 4).

17. Write a C program using a for loop to calculate the factorial of a given number. in preetry modrate word , it is for both understanding and exam writing

```
#include <stdio.h> // Required for input/output functions like printf and scanf
int main() {
  // Declare variables
  int num;       // To store the number entered by the user
  long long factorial = 1; // To store the calculated factorial.
                  // Using 'long long' to handle larger factorials,
                  // as factorials grow very quickly.
  int i;        // Loop counter variable
```

```c
    // Prompt the user to enter a non-negative integer
    printf("Enter a non-negative integer: ");
    scanf("%d", &num); // Read the integer entered by the user

    // --- Input Validation and Edge Cases ---
    // Factorial is not defined for negative numbers
    if (num < 0) {
        printf("Error: Factorial is not defined for negative numbers.\n");
    }
    // Factorial of 0 is 1 by definition
    else if (num == 0) {
        printf("The factorial of 0 is: 1\n");
    }
    // --- Calculate Factorial for positive numbers using a for loop ---
    else {
        // The for loop iterates from 1 up to 'num' (inclusive)
        for (i = 1; i <= num; i++) {
            factorial *= i; // This is equivalent to: factorial = factorial * i;
                            // In each iteration, 'factorial' is updated by
                            // multiplying its current value with the loop counter 'i'.
        }
        // Print the calculated factorial
        printf("The factorial of %d is: %lld\n", num, factorial);
    }
    return 0; // Indicate successful program execution
}
```

18. Write a C program to check whether a given number is positive, negative, or zero using if-else statements.

```c
#include <stdio.h> // Include the standard input/output library for printf and scanf

int main() {

  // Declare an integer variable to store the number entered by the user

  int number;



  // Prompt the user to enter an integer

  printf("Enter an integer: ");

  // Read the integer from the user and store it in the 'number' variable

  scanf("%d", &number);

  // --- Conditional Logic using if-else if-else ---

  // Check if the number is positive

  if (number > 0) {

    printf("%d is a positive number.\n", number);

  }

  // If the number is not positive, check if it is negative

  else if (number < 0) {

    printf("%d is a negative number.\n", number);

  }

  // If the number is neither positive nor negative, it must be zero

  else { // This 'else' block executes if 'number == 0'

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

  }

  // Indicate successful program execution
```

```
  return 0;

}
```

19. Explain the working of the switch statement in C with an example.

The switch statement in C is a control flow statement that allows you to execute different blocks of code based on the value of a single variable or expression. It's often used as an alternative to a long chain of if-else if-else statements when you have multiple possible execution paths depending on a specific value.

It's particularly useful when you need to perform different actions based on different integer or character values.

How the switch Statement Works:

1. **Expression Evaluation**: The switch statement evaluates an expression (which must result in an integer type, like int, char, enum, or long).

2. **Case Matching**: The result of this expression is then compared sequentially with the values provided in case labels.

3. **Code Execution**:

    - If a case value matches the expression's result, the code block associated with that case is executed.

    - **break Statement (Crucial!)**: After the code for a matching case is executed, a break statement is typically used. The break statement immediately exits the switch block, and program control continues with the statement following the switch.

    - **"Fall-through"**: If break is omitted, execution "falls through" to the next case label (and subsequent ones) and executes their code blocks as well, until a break is encountered or the switch block ends. This "fall-through" behavior is sometimes intentional but often leads to bugs if not handled carefully.

4. **default Case (Optional)**: If none of the case values match the expression's result, the code block under the default label (if present) is executed. The default case acts like the final else in an if-else if chain. It's good practice to include a default case for handling unexpected or unhandled values.

Syntax of switch Statement:

```
switch (expression) {

  case constant_value_1:

    // Code to execute if expression matches constant_value_1
```

```
    break; // Exits the switch statement
  case constant_value_2:
    // Code to execute if expression matches constant_value_2
    break; // Exits the switch statement
  // ... more case statements ...
  default:
    // Code to execute if no case matches (optional)
    // No break needed here as it's the last block
}
```

- expression: An integer or character expression.
- constant_value_N: Must be a constant integer or character value. Variables are not allowed here.

Example: Simple Calculator using switch

Let's create a C program that acts as a very basic calculator, performing addition, subtraction, multiplication, or division based on the user's choice (an operator character).

```
#include <stdio.h> // Include standard input/output library for printf and scanf
int main() {
  char operator;   // To store the operator character (+, -, *, /)
  double num1, num2; // To store the two numbers for calculation (using double for decimal support)

  // Prompt the user to enter the operator
  printf("Enter an operator (+, -, *, /): ");
  scanf(" %c", &operator); // Note the space before %c to consume any leftover newline character
                           // from previous inputs, preventing issues.
  // Prompt the user to enter two numbers
  printf("Enter two operands: ");
  scanf("%lf %lf", &num1, &num2); // %lf is used for reading double values
```

```
    // Use the switch statement to perform operations based on the operator
    switch (operator) {
        case '+': // If operator is '+'
            printf("%.2lf + %.2lf = %.2lf\n", num1, num2, num1 + num2);
            break; // Exit the switch statement
        case '-': // If operator is '-'
            printf("%.2lf - %.2lf = %.2lf\n", num1, num2, num1 - num2);
            break; // Exit the switch statement
        case '*': // If operator is '*'
            printf("%.2lf * %.2lf = %.2lf\n", num1, num2, num1 * num2);
            break; // Exit the switch statement
        case '/': // If operator is '/'
            // Check for division by zero to prevent runtime errors
            if (num2 != 0) {
                printf("%.2lf / %.2lf = %.2lf\n", num1, num2, num1 / num2);
            } else {
                printf("Error: Division by zero is not allowed.\n");
            }
            break; // Exit the switch statement
        default: // If the operator does not match any of the above cases
            printf("Error: Invalid operator entered.\n");
            // No break needed here as it's the last block in the switch
    }
    return 0; // Indicate successful program execution
}
```

20. What is a selection statement and explain its types briefly?

A **selection statement** (also known as a **conditional statement** or **decision control statement**) is a fundamental programming construct that allows a program to make decisions and execute different blocks of code based on whether a specified condition evaluates to true or false. In essence, it enables a program to choose which path of execution to follow.

Without selection statements, a program would execute instructions strictly sequentially, from top to bottom, every single time. Selection statements introduce "intelligence" by allowing the program to react to different inputs, states, or circumstances.

Types of Selection Statements:

In C (and many other programming languages), the primary types of selection statements are:

1. **if statement:**
    - **Purpose:** The simplest form of selection. It executes a block of code *only if* a specified condition is true. If the condition is false, the block is skipped, and the program continues with the statements immediately following the if block.
    - **Brief Explanation:** It's like saying, "IF this is true, THEN do this."
    - **Syntax (Conceptual):**

```
if (condition) {
   // Code to execute if condition is true
}
// Program continues here
```

    - **if-else statement: Purpose:** Provides two possible paths of execution. One block of code is executed if the condition is true, and a different block of code is executed if the condition is false.
    - **Brief Explanation:** It's like saying, "IF this is true, THEN do this; OTHERWISE (ELSE), do that instead."
    - **Syntax (Conceptual):**

```
if (condition) {
   // Code to execute if condition is true
} else {
   // Code to execute if condition is false
```

```
}

// Program continues here
```

2. **if-else if-else statement (or if-else if ladder):**

    - **Purpose:** Used when there are multiple conditions to check, and each condition might lead to a different action. It allows you to test a series of conditions sequentially. The first condition that evaluates to true has its corresponding code block executed, and then the rest of the else if and else blocks are skipped. If none of the if or else if conditions are true, the final else block (if present) is executed.

    - **Brief Explanation:** It's like saying, "IF this is true, do A; ELSE IF that is true, do B; ELSE IF something else is true, do C; OTHERWISE (ELSE), do D."

    - **Syntax (Conceptual):**

```
if (condition1) {

   // Code if condition1 is true

} else if (condition2) {

   // Code if condition1 is false AND condition2 is true

} else if (condition3) {

   // Code if condition1 and condition2 are false AND condition3 is true

} else {

   // Code if all preceding conditions are false (optional)

}

// Program continues here
```

3. **switch statement:**

    - **Purpose:** Designed for situations where you need to choose among many different code blocks based on the *value* of a single variable or expression. It's often a more readable and efficient alternative to a long if-else if-else ladder when dealing with discrete, constant values (like integer codes, characters, or enumerated types).

    - **Brief Explanation:** It evaluates an Texpression and then jumps to the case label whose value matches the expression's result. The break statement is crucial to prevent "fall-through" to subsequent cases. An optional default case handles values that don't match any case.

- **Syntax (Conceptual):**

```
switch (expression) {
  case constant_value_1:
    // Code for value 1
    break;
  case constant_value_2:
    // Code for value 2
    break;
  // ... more cases ...
  default:
    // Code if no match (optional)
}
// Program continues here
```

21A. What is a Table of Strings? How to Declare and Initialize it?

In C, a string is fundamentally an array of characters terminated by a null character (\0). A "table of strings" is essentially a collection of these individual strings. There are primarily two common ways to represent and handle a table of strings in C:

1. **Using a 2D Character Array (char array_name[rows][columns];)**
2. **Using an Array of Character Pointers (char *array_name[];)**

Let's look at each:

1. Using a 2D Character Array (char array_name[rows][columns];)

- **What it is:** This method creates a fixed-size block of memory where each "row" can store one string. You define the maximum number of strings (rows) and the maximum length of each string (including the null terminator, columns).
- **Declaration:**

```
char names[5][20]; // Declares an array to hold 5 strings,
                   // each string can have a maximum of 19 characters + '\0'
```

- **Initialization:**
    - **At Declaration (Recommended for fixed strings):**

```
char colors[3][10] = {
  "Red",   // string 0 (max 9 chars + '\0')
  "Green",  // string 1
  "Blue"   // string 2
};
// The compiler automatically figures out '3' if you omit it:
// char colors[][10] = {"Red", "Green", "Blue"};
```

    - **After Declaration (Using strcpy()):** You **cannot** use the assignment operator (=) directly with character arrays after declaration. You must use string manipulation functions like strcpy() from the <string.h> library.

```
#include <string.h> // For strcpy
char fruits[3][15]; // Declare a 2D array for 3 fruits, max 14 chars each
strcpy(fruits[0], "Apple");
strcpy(fruits[1], "Banana");
strcpy(fruits[2], "Cherry");
```

- **Pros:** Simple for fixed-length strings, memory is contiguous.
- **Cons:** Can waste memory if strings are much shorter than the columns size. Fixed maximum length for each string.

2. Using an Array of Character Pointers (char *array_name[];)

- **What it is:** This method creates an array where each element is a pointer (char *). Each pointer then points to the first character of a string, which can be stored elsewhere in memory (often

as string literals, which are read-only). This is generally more memory-efficient when strings have varying lengths.

- **Declaration:**

```
char *messages[4]; // Declares an array of 4 character pointers
```

- **Initialization:**
    - **At Declaration (Using String Literals - Most Common):**

```
char *days[] = { // The compiler automatically figures out the size (7 in this case)
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
  "Sunday"
};
// Each element (e.g., days[0]) is a pointer to the start of the string "Monday".
// Note: String literals are typically stored in read-only memory.
```

    - **After Declaration (Dynamically using malloc and strcpy - More Advanced):** For strings that need to be modified or created at runtime, you'd dynamically allocate memory for each string and copy content into it.

```
#include <stdlib.h> // For malloc
#include <string.h> // For strcpy


char *words[2]; // Declare an array of 2 character pointers
words[0] = (char *)malloc(sizeof(char) * 10); // Allocate memory for 9 chars + '\0'
strcpy(words[0], "Hello");
words[1] = (char *)malloc(sizeof(char) * 15);
```

```
strcpy(words[1], "World");

// Remember to free allocated memory when done: free(words[0]); free(words[1]);
```

- **Pros:** More memory efficient for strings of varying lengths. Flexible as strings can be of different sizes.
- **Cons:** Strings are often read-only (if using string literals). More complex for dynamic allocation.

21B. C Program to Initialize and Display the Contents of a Table of Strings

This program will demonstrate both methods: initializing a 2D character array and an array of character pointers, and then displaying their contents.

```
#include <stdio.h>  // Required for printf

#include <string.h> // Required for strcpy (if using 2D array and initializing after declaration)

int main() {

  // --- Method 1: Using a 2D Character Array ---

  printf("--- Method 1: Using a 2D Character Array ---\n");

  // Declaration and initialization at the same time

  // Max 4 strings, each up to 9 characters + null terminator

  char programming_languages[4][10] = {

    "C",

    "Python",

    "Java",

    "Go"

  };

  // To get the number of strings, divide the total size of the array

  // by the size of one row (i.e., the size of one string).

  int num_languages = sizeof(programming_languages) / sizeof(programming_languages[0]);

  int i; // Loop counter
```

```
printf("Programming Languages:\n");
for (i = 0; i < num_languages; i++) {
    printf("  %d. %s\n", i + 1, programming_languages[i]);
}
printf("\n");
// --- Method 2: Using an Array of Character Pointers ---
printf("--- Method 2: Using an Array of Character Pointers ---\n");
// Declaration and initialization (using string literals)
char *planets[] = {
    "Mercury",
    "Venus",
    "Earth",
    "Mars",
    "Jupiter",
    "Saturn",
    "Uranus",
    "Neptune"
};
// To get the number of strings, divide the total size of the pointer array
// by the size of a single pointer (char*).
int num_planets = sizeof(planets) / sizeof(planets[0]);
printf("Planets in our Solar System:\n");
for (i = 0; i < num_planets; i++) {
    printf("  %d. %s\n", i + 1, planets[i]);
}
```

```
    printf("\n");

    return 0; // Indicate successful program execution

}
```

22. Write a C program to find the sum of all even and odd numbers separately from 1 to N, where N is entered by the user. Explain the logic used in your program.

```
#include <stdio.h> // Include the standard input/output library for printf and scanf

int main() {

    // Declare variables

    int N;          // To store the upper limit entered by the user

    int sum_even = 0;  // To store the sum of even numbers, initialized to 0

    int sum_odd = 0;   // To store the sum of odd numbers, initialized to 0

    int i;          // Loop counter variable

    // Prompt the user to enter the upper limit (N)

    printf("Enter a positive integer (N): ");

    scanf("%d", &N); // Read the integer entered by the user



    // --- Input Validation ---

    // Ensure N is a positive integer for the loop to make sense

    if (N <= 0) {

        printf("Error: Please enter a positive integer for N.\n");

        return 1; // Indicate an error and exit the program

    }

    // --- Logic to calculate sums using a for loop ---

    // The loop iterates through each number from 1 up to N (inclusive)
```

```
  for (i = 1; i <= N; i++) {
    // Check if the current number 'i' is even or odd
    // An even number has a remainder of 0 when divided by 2
    if (i % 2 == 0) {
      sum_even += i; // If 'i' is even, add it to sum_even
    } else {
      sum_odd += i;  // If 'i' is odd, add it to sum_odd
    }
  }
  // --- Display the results ---
  printf("\n--- Results ---\n");
  printf("Sum of even numbers from 1 to %d: %d\n", N, sum_even);
  printf("Sum of odd numbers from 1 to %d: %d\n", N, sum_odd);
  return 0; // Indicate successful program execution
}
```

Logic Used in the Program:

1. **Initialization**:
    - int N;: A variable N is declared to store the upper limit provided by the user.
    - int sum_even = 0;: A variable sum_even is initialized to 0. This variable will accumulate the sum of all even numbers found within the range. It's crucial to initialize it to 0 so that the first addition is correct.
    - int sum_odd = 0;: Similarly, sum_odd is initialized to 0 to accumulate the sum of all odd numbers.
    - int i;: A loop counter variable i is declared.
2. **User Input**:
    - printf("Enter a positive integer (N): "); prompts the user to enter a value for N.

- scanf("%d", &N); reads the integer entered by the user and stores it in the N variable.

3. **Input Validation**:
   - if (N <= 0): This if statement checks if the entered N is less than or equal to 0. Since the problem asks for sums from 1 to N, N must be a positive integer. If it's not, an error message is printed, and the program exits. This prevents the loop from running incorrectly or infinitely for invalid inputs.
4. **Iterating and Summing (The for loop)**:
   - **for (i = 1; i <= N; i++) { ... }**: This for loop is the core of the logic.
     - **i = 1**: The loop starts checking numbers from 1.
     - **i <= N**: The loop continues as long as the current number i is less than or equal to the user-provided N. This ensures that all numbers in the range [1, N] are considered.
     - **i++**: After each iteration, i is incremented by 1, moving to the next number in the sequence.
5. **Checking Even/Odd (if-else statement)**:
   - **if (i % 2 == 0)**: Inside the loop, for each number i, this condition checks if i is even.
     - The modulo operator (%) gives the remainder of a division. If a number i divided by 2 has a remainder of 0, it means i is an even number.
     - If i is even, sum_even += i; adds the current number i to the sum_even variable.
   - **else { ... }**: If the if condition (i % 2 == 0) is false, it means i is not even, and therefore it must be an odd number.
     - In this else block, sum_odd += i; adds the current number i to the sum_odd variable.
6. **Displaying Results**:
   - After the for loop completes its execution (i.e., i has gone past N), printf statements are used to display the final calculated sum_even and sum_odd to the user.

This logic ensures that every number from 1 to N is checked once, correctly categorized as even or odd, and added to its respective sum, providing an accurate result.

23. Write a C program that takes the total number of classes held and the number of classes attended as input. The program should check if the student is eligible for exams using conditional statements. Explain the logic used for decision-making. (Note:- A school requires students to attend at least 75% of classes to be eligible for exams.)

```c
#include <stdio.h> // Include standard input/output library for printf and scanf

int main() {

  // Declare variables

  int total_classes_held;   // To store the total number of classes conducted

  int classes_attended;     // To store the number of classes the student attended

  double attendance_percentage;

// To store the calculated attendance percentage (using double for precision)

  // --- Input Phase ---

  // Prompt the user for the total number of classes held

  printf("Enter the total number of classes held: ");

  scanf("%d", &total_classes_held); // Read the input

  // Prompt the user for the number of classes attended

  printf("Enter the number of classes attended by the student: ");

  scanf("%d", &classes_attended); // Read the input

  // --- Input Validation (Important for robust programs) ---

  // Check for invalid scenarios before calculation

  if (total_classes_held <= 0) {

    printf("Error: Total classes held must be a positive number.\n");

    return 1; // Indicate an error and exit

  }

  if (classes_attended < 0) {

    printf("Error: Classes attended cannot be negative.\n");
```

```
    return 1; // Indicate an error and exit
  }
  if (classes_attended > total_classes_held) {
    printf("Error: Classes attended cannot be more than total classes held.\n");
    return 1; // Indicate an error and exit
  }
  // --- Calculation Phase ---
  // Calculate the attendance percentage
  // Cast one of the integers to double to ensure floating-point division
  attendance_percentage = ((double)classes_attended / total_classes_held) * 100.0;
  // --- Decision-Making Phase (Conditional Statements) ---
  printf("\n--- Eligibility Check ---\n");
  printf("Total Classes Held: %d\n", total_classes_held);
  printf("Classes Attended: %d\n", classes_attended);
  printf("Attendance Percentage: %.2lf%%\n", attendance_percentage); // Display percentage with 2 decimal places

  // Check if the attendance percentage meets the eligibility criteria (75%)
  if (attendance_percentage >= 75.0) {
    printf("Result: Congratulations! You are eligible to sit for the exams.\n");
  } else {
    printf("Result: Sorry, you are NOT eligible to sit for the exams. Your attendance is below 75%%.\n");
  }
  return 0; // Indicate successful program execution
}
```

Logic Used for Decision-Making:

The program's decision-making logic revolves around a few key steps:

1. **Gathering Inputs:**
    - The program first requests two crucial pieces of information from the user:
        - total_classes_held: The total number of classes that were conducted.
        - classes_attended: The actual number of classes the student participated in.
2. **Input Validation:**
    - Before performing any calculations, the program includes if statements to validate the user's input. This is a critical step for robust programs.
        - It checks if total_classes_held is zero or negative. Division by zero is mathematically undefined and would cause a runtime error.
        - It checks if classes_attended is negative (which makes no sense).
        - It checks if classes_attended is greater than total_classes_held (which is impossible).
    - If any of these invalid conditions are met, an error message is displayed, and the program exits gracefully using return 1; (indicating an error).
3. **Calculating Attendance Percentage:**
    - attendance_percentage = ((double)classes_attended / total_classes_held) * 100.0;
    - This is the core calculation. To ensure accurate floating-point division (which is necessary for percentages), one of the integer variables (classes_attended) is explicitly cast to a double using (double). If this cast were omitted, integer division would occur, potentially leading to incorrect percentages (e.g., 3/4 would be 0 instead of 0.75). The result is then multiplied by 100.0 to convert it into a percentage.
4. **Conditional Decision (if-else statement):**
    - if (attendance_percentage >= 75.0): This is the primary conditional statement that makes the eligibility decision.
        - **Condition:** It compares the calculated attendance_percentage with the required minimum of 75.0.
        - **If True:** If the student's attendance percentage is 75.0 or higher, the code inside the if block is executed. This prints a message indicating that the student is eligible for the exams.

- **If False:** If the condition attendance_percentage >= 75.0 is false (meaning the percentage is less than 75.0), the code inside the else block is executed. This prints a message stating that the student is *not* eligible.

24. Compare and contrast the if-else statement and the switch statement in C. Write a program that demonstrates the use of both for solving the same problem.

Comparison and Contrast: if-else vs. switch in C

Both if-else and switch are **selection statements** (or conditional statements) in C, meaning they allow your program to make decisions and execute different blocks of code based on conditions. However, they are suited for different types of decision-making scenarios.

Similarities:

1. **Decision Making:** Both allow the program to choose one path of execution out of several possible paths.
2. **Control Flow:** They alter the sequential flow of a program based on certain conditions.
3. **Default/Catch-all:** Both can handle situations where no specific condition is met (else for if-else, default for switch).

Differences:

Feature	if-else Statement	switch Statement
Expression Type	Evaluates a boolean expression (true/false). Can handle complex logical conditions (&&, `	,!`), ranges, and floating-point comparisons.
Conditions	Ideal for checking a range of values or complex conditions involving multiple variables and logical operators.	Ideal for checking against multiple discrete, constant values. Each case must be a unique constant.
Readability	Can become less readable (a "ladder") if there are many else if conditions, especially for simple value checks.	Often more readable and structured when dealing with a large number of discrete, single-value comparisons.
Execution Flow	Executes the first if or else if block whose condition is true, then skips the rest of the ladder. No "fall-through."	Executes the case block that matches the expression. Requires break to exit the switch block; otherwise, execution

		"falls through" to subsequent case blocks.
Flexibility	More flexible; can handle any type of condition.	Less flexible; restricted to integer-type expressions and constant case values.
Performance	For a few conditions, performance difference is negligible. For many discrete conditions, switch can sometimes be optimized by the compiler for faster jumps (e.g., using jump tables).	Can be slightly more efficient for a large number of discrete cases due to compiler optimizations.

Program Demonstrating Both for the Same Problem

```
#include <stdio.h> // Include standard input/output library for printf and scanf

int main() {

  int day_number; // To store the day number entered by the user

  // --- Using if-else if-else statement ---

  printf("--- Using if-else if-else statement ---\n");

  printf("Enter a day number (1-7): ");

  scanf("%d", &day_number);

  if (day_number == 1) {

    printf("Day: Monday\n");

  } else if (day_number == 2) {

    printf("Day: Tuesday\n");

  } else if (day_number == 3) {

    printf("Day: Wednesday\n");

  } else if (day_number == 4) {

    printf("Day: Thursday\n");

  } else if (day_number == 5) {

    printf("Day: Friday\n");
```

```
} else if (day_number == 6) {
    printf("Day: Saturday\n");
} else if (day_number == 7) {
    printf("Day: Sunday\n");
} else {
    printf("Error: Invalid day number. Please enter a number between 1 and 7.\n");
}
printf("\n"); // Add a newline for better separation
// --- Using switch statement ---
printf("--- Using switch statement ---\n");
printf("Enter a day number (1-7): ");
scanf("%d", &day_number); // Read input again for the switch example
switch (day_number) {
    case 1:
        printf("Day: Monday\n");
        break; // Important: Exits the switch block
    case 2:
        printf("Day: Tuesday\n");
        break;
    case 3:
        printf("Day: Wednesday\n");
        break;
    case 4:
        printf("Day: Thursday\n");
        break;
    case 5:
```

```
        printf("Day: Friday\n");
        break;
    case 6:
        printf("Day: Saturday\n");
        break;
    case 7:
        printf("Day: Sunday\n");
        break;
    default: // This case executes if day_number doesn't match any of the above
        printf("Error: Invalid day number. Please enter a number between 1 and 7.\n");
        // No break needed here as it's the last block
  }
  return 0; // Indicate successful program execution
}
```

25. What are different types of pointers?Explain briefly.

In C, pointers are variables that store memory addresses. The "type" of a pointer refers to the data type of the variable it points to. This type information is crucial because it tells the compiler how to interpret the data at the memory address and how many bytes to access.

Here are the different types of pointers, explained briefly:

1. **Null Pointers:**
    - A null pointer is a pointer that points to nothing, or to a memory location that cannot be accessed. It's typically represented by the NULL macro (which is usually defined as (void*)0 or 0).
    - **Purpose:** Used to initialize pointers when they don't point to a valid memory location yet, or to indicate that a pointer variable is intentionally not pointing to any object. Dereferencing a null pointer leads to undefined behavior (often a program crash).
    - **Example:** int *ptr = NULL;

2. **Void Pointers (Generic Pointers):**
    - A void pointer (void *) is a generic pointer that can hold the address of any data type. It's often called a "generic" pointer because it doesn't have a specific type associated with the data it points to.
    - **Purpose:** Useful for functions that handle data of different types (e.g., malloc(), calloc(), memcpy()).
    - **Limitation:** You cannot directly dereference a void pointer. You must first cast it to a specific data type pointer before dereferencing it. Pointer arithmetic is also not directly allowed on void pointers because the compiler doesn't know the size of the data type it points to.
    - **Example:**

```
int x = 10;

void *ptr = &x;

// printf("%d", *ptr); // ERROR: cannot dereference void*

printf("%d", *(int*)ptr); // Correct: cast to int* then dereference
```

3. **Wild Pointers:**
    - A wild pointer is a pointer that has been declared but not initialized to point to a valid memory address (or NULL). It contains a garbage value (an arbitrary memory address).
    - **Danger:** Dereferencing a wild pointer leads to undefined behavior, as it might point to an invalid or unintended memory location, potentially causing crashes or corrupting data.
    - **Example:** int *ptr; // ptr is a wild pointer here
4. **Dangling Pointers:**
    - A dangling pointer is a pointer that points to a memory location that has been deallocated (freed) or no longer exists. The memory might now be used by another part of the program or the operating system.
    - **Danger:** Dereferencing a dangling pointer also leads to undefined behavior, as you're accessing memory that you no longer "own."
    - **Common Causes:**
        - Freeing memory pointed to by a pointer (free(ptr);).
        - Returning the address of a local variable from a function (the local variable's memory is deallocated when the function returns).

        - When a block of memory is deallocated, but other pointers still point to it.
    - **Example:**

```
int *ptr = (int *)malloc(sizeof(int));

// ... use ptr ...

free(ptr); // Memory is freed, ptr is now dangling

ptr = NULL; // Good practice: set to NULL after freeing
```

5. **Function Pointers:**
    - A function pointer is a pointer that stores the memory address of a function. This allows you to call a function indirectly through the pointer.
    - **Purpose:** Useful for implementing callbacks, dispatch tables, or designing flexible and extensible code (e.g., passing a comparison function to a sorting algorithm).
    - **Example:** int (*func_ptr)(int, int); // Declares a pointer to a function that takes two ints and returns an int

26. Differentiate between a one-dimensional (1-D) and a two-dimensional (2-D) array with examples.

Arrays in programming are fundamental data structures used to store collections of elements of the same data type. The distinction between one-dimensional (1-D) and two-dimensional (2-D) arrays lies in their structure and how elements are organized and accessed.

1. One-Dimensional (1-D) Array

A **one-dimensional array** is the simplest form of an array, representing a linear list or a sequence of elements. Think of it as a single row or a column where elements are stored contiguously in memory, one after another. Each element is identified by a single subscript (or index).

- **Analogy:** A shopping list, a list of student scores, or a single row of books on a shelf.
- **Structure:** A single row of data.
- **Declaration Syntax:**

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

    - data_type: The type of elements (e.g., int, float, char).
    - array_name: The name of the array.

- size: The number of elements the array can hold.

- **Memory Representation:** Elements are stored in a single, continuous block of memory.
- **Accessing Elements:** You access an element using a single index, starting from 0 for the first element up to size-1 for the last element.

```
array_name[index];
```

Example (C Program):

```c
#include <stdio.h>

int main() {

  // Declaration and Initialization of a 1-D array

  int scores[5] = {85, 92, 78, 95, 88}; // An array to store 5 integer scores

  printf("--- One-Dimensional Array Example ---\n");

  printf("Scores of 5 students:\n");


  // Accessing and displaying elements using a single loop

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

    printf("Score at index %d: %d\n", i, scores[i]);

  }

  // You can also modify an element

  scores[2] = 80; // Change the score at index 2 from 78 to 80

  printf("Updated score at index 2: %d\n", scores[2]);

  return 0;

}
```

2. Two-Dimensional (2-D) Array

A **two-dimensional array** can be thought of as an "array of arrays." It represents a grid or a table of elements, organized into rows and columns. While conceptually it has two dimensions, in memory, elements are still stored contiguously, usually in row-major order (all elements of the first row, then all elements of the second row, and so on).

- **Analogy:** A spreadsheet, a chessboard, a matrix in mathematics, or a seating arrangement in a classroom.
- **Structure:** Data organized in rows and columns.
- **Declaration Syntax:**

```
data_type array_name[rows][columns];
```

    - rows: The number of rows in the table.
    - columns: The number of columns in the table.

- **Memory Representation:** Though conceptually a grid, in memory, it's a linear block. C stores 2D arrays in **row-major order**, meaning all elements of the first row are stored consecutively, followed by all elements of the second row, and so on.
- **Accessing Elements:** You access an element using two subscripts: one for the row index and one for the column index. Both indices typically start from 0.

```
array_name[row_index][column_index];
```

Example (C Program):

```
#include <stdio.h>

int main() {

  // Declaration and Initialization of a 2-D array (a 3x3 matrix)

  int matrix[3][3] = {

    {1, 2, 3},   // Row 0

    {4, 5, 6},   // Row 1

    {7, 8, 9}    // Row 2

  };

  printf("\n--- Two-Dimensional Array Example ---\n");

  printf("Matrix (3x3):\n");

  // Accessing and displaying elements using nested loops

  // Outer loop for rows

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

```
    // Inner loop for columns
    for (int j = 0; j < 3; j++) {
        printf("%d ", matrix[i][j]); // Print element followed by a space
    }
    printf("\n"); // Move to the next line after printing all elements of a row
  }
  // You can also modify an element
  matrix[1][1] = 10; // Change element at row 1, column 1 (which was 5) to 10
  printf("Updated element at matrix[1][1]: %d\n", matrix[1][1]);
  return 0;
}
```

Feature	One-Dimensional (1-D) Array	Two-Dimensional (2-D) Array
Structure	Linear list of elements; a single row/column	Table/grid of elements; organized in rows and columns
Indexing	Requires a single index/subscript	Requires two indices/subscripts (row, column)
Declaration	data_type array_name[size];	data_type array_name[rows][columns];
Conceptual View	A list, a vector	A table, a matrix
Memory Access	array_name[index]	array_name[row_index][column_index]
Use Cases	Storing lists of items, sequences	Storing tabular data, matrices, images

27. Write a C program to find the largest element in a 1-D array.

```c
#include <stdio.h> // Include standard input/output library for printf and scanf

int main() {

  // Declare variables

  int n; // To store the number of elements in the array

  int arr[100]; // Declare a 1-D integer array with a maximum size of 100 elements

            // (You can adjust this size as needed)

  int largest_element; // To store the largest element found in the array

  int i; // Loop counter

  // --- Input Phase ---

  // Prompt the user to enter the number of elements

  printf("Enter the number of elements (1-100): ");

  scanf("%d", &n); // Read the number of elements


  // Input Validation: Ensure 'n' is within a reasonable range

  if (n <= 0 || n > 100) {

    printf("Error: Please enter a number of elements between 1 and 100.\n");

    return 1; // Indicate an error and exit the program

  }

  // Prompt the user to enter the array elements

  printf("Enter %d integer elements:\n", n);

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

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

    scanf("%d", &arr[i]); // Read each element into the array

  }
```

```c
    // --- Logic to Find the Largest Element ---
    // Assume the first element is initially the largest
    // This is a common starting point for comparison
    largest_element = arr[0];
    // Loop through the rest of the array elements, starting from the second element (index 1)
    for (i = 1; i < n; i++) {
        // Compare the current element with the 'largest_element' found so far
        if (arr[i] > largest_element) {
            largest_element = arr[i]; // If the current element is larger, update 'largest_element'
        }
    }
    // --- Output Phase ---
    printf("\n--- Result ---\n");
    printf("The largest element in the array is: %d\n", largest_element);

    return 0; // Indicate successful program execution
}
```

28. Write a C program to implement the Linear Search algorithm.

```c
#include <stdio.h>
// Function to implement Linear Search
// array: The array to search in
// size: The number of elements in the array
// key: The element to search for
int linear_search(int array[], int size, int key) {
```

```
    // Iterate through each element of the array
    for (int i = 0; i < size; i++) {
        // If the current element matches the key
        if (array[i] == key) {
            return i; // Return the index where the key was found
        }
    }
    // If the loop completes without finding the key,
    // it means the key is not present in the array.
    return -1; // Return -1 to indicate that the element was not found
}
int main() {
    int n; // Number of elements in the array
    int arr[100]; // Array declaration
    int search_key; // Element to search for
    int result_index; // To store the result of the search
    printf("Enter the number of elements in the array (1-100): ");
    scanf("%d", &n);
    // Input validation for array size
    if (n <= 0 || n > 100) {
        printf("Invalid number of elements. Please enter a value between 1 and 100.\n");
        return 1;
    }
    printf("Enter %d integer elements:\n", n);
    for (int i = 0; i < n; i++) {
        printf("Element %d: ", i + 1);
```

```
    scanf("%d", &arr[i]);
  }
  printf("Enter the element to search for: ");
  scanf("%d", &search_key);
  // Call the linear_search function
  result_index = linear_search(arr, n, search_key);
  // Display the result
  if (result_index != -1) {
    printf("Element %d found at index %d.\n", search_key, result_index);
    printf("Position (1-based): %d\n", result_index + 1);
  } else {
    printf("Element %d not found in the array.\n", search_key);
  }
  return 0;
}
```

29. What is recursion? Explain with an example. Compare recursion and iteration.

What is Recursion?

Recursion is a programming technique where a function calls itself, either directly or indirectly, to solve a problem. Think of it as a problem-solving approach where a complex problem is broken down into smaller, identical sub-problems until a simple, solvable base case is reached.

Every recursive function must have two main parts:

1. **Base Case:** This is the condition that stops the recursion. Without a base case, the function would call itself indefinitely, leading to an infinite loop (and eventually a stack overflow error). The base case provides a direct solution for the simplest form of the problem.

2. **Recursive Step:** This is where the function calls itself with a modified (usually smaller or simpler) version of the original problem. The idea is that each recursive call moves closer to the base case.

Example: Calculating Factorial using Recursion

The factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n. For example, 5!=5×4×3×2×1=120. By definition, 0!=1.

We can define factorial recursively:

- **Base Case:** If n=0, then n!=1.
- **Recursive Step:** If n>0, then n!=n×(n−1)!.

```
#include <stdio.h> // For input/output functions

// Function to calculate factorial using recursion

long long factorial(int n) {

  // Base Case: If n is 0, return 1. This stops the recursion.

  if (n == 0) {

    return 1;

  }

  // Recursive Step: If n is greater than 0,

  // return n multiplied by the factorial of (n-1).

  else {

    return n * factorial(n - 1);

  }

}

int main() {

  int num;

  long long result;

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

  scanf("%d", &num);

  // Input validation

  if (num < 0) {
```

```
    printf("Error: Factorial is not defined for negative numbers.\n");
  } else {
    result = factorial(num);
    printf("Factorial of %d is: %lld\n", num, result);
  }
  return 0;
}
```

Definition	A function calls itself.	A block of code is repeated using loops.
Control Flow	Managed by function call stack (LIFO). Each recursive call adds a new stack frame.	Managed by loop control variables and conditions.
Termination	Requires a base case to stop.	Requires a loop condition to become false.
Memory Usage	Can consume more memory (stack space) due to multiple function calls being pushed onto the call stack. Can lead to "Stack Overflow" for deep recursions.	Generally more memory-efficient as it doesn't involve multiple function calls and stack frames.
Code Complexity	Can lead to more elegant and concise code for problems that have a natural recursive definition (e.g., tree traversals, certain mathematical functions).	Can be more verbose for problems naturally defined recursively. Simpler for straightforward repetitive tasks.
Performance	Generally slower due to overhead of function calls (stack frame creation, saving/restoring registers).	Generally faster as it avoids function call overhead.
Readability	Can be harder to trace and debug for beginners due to the call stack.	Easier to understand and trace the flow of execution.

When to Use	When the problem has a recursive structure (e.g., fractals, tree/graph algorithms, divide-and-conquer algorithms). When elegance and conciseness outweigh minor performance differences.	When the problem involves simple repetition or counting. When performance and memory efficiency are critical. When the problem doesn't have an obvious recursive structure.

30. Explain the working of the Bubble Sort algorithm with an example.

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until no swaps are needed, indicating that the list is sorted. The name "bubble sort" comes from the way smaller (or larger, depending on sort order) elements "bubble" to their correct positions.

Core Idea: In each pass (iteration) through the list, the largest unsorted element "bubbles up" to its correct sorted position at the end of the unsorted portion of the array.

Steps (for ascending order):

1. **Start from the beginning** of the list.
2. **Compare adjacent elements:** Look at the current element and the one next to it.
3. **Swap if necessary:** If the current element is greater than the next element, swap them.
4. **Move to the next pair:** Increment the index and repeat steps 2-3 until the end of the unsorted portion of the list is reached.
5. **One pass completed:** After one complete pass, the largest unsorted element will be in its correct sorted position at the end of the array.
6. **Repeat:** Repeat steps 1-5 for the remaining unsorted portion of the list. The unsorted portion shrinks by one element after each pass.
7. **Termination:** The algorithm terminates when a full pass occurs with no swaps, meaning the array is sorted. Alternatively, it can run for n-1 passes, where n is the number of elements, guaranteed to sort the array.

Example: Sorting [5, 1, 4, 2, 8] in Ascending Order

Let's trace the Bubble Sort algorithm with the array arr = [5, 1, 4, 2, 8]. Number of elements n = 5.

Pass 1: (Largest element 8 bubbles to the end) Compare (arr[0], arr[1]) -> (5, 1):

- 5 > 1, so swap. arr becomes [1, 5, 4, 2, 8]

Compare (arr[1], arr[2]) -> (5, 4):

- 5 > 4, so swap. arr becomes [1, 4, 5, 2, 8]

Compare (arr[2], arr[3]) -> (5, 2):

- 5 > 2, so swap. arr becomes [1, 4, 2, 5, 8]

Compare (arr[3], arr[4]) -> (5, 8):

- 5 is NOT > 8, no swap. arr remains [1, 4, 2, 5, 8]

End of Pass 1. The largest element 8 is now at its correct position. The sorted part is [8]. The unsorted part is [1, 4, 2, 5].

Pass 2: (Largest remaining unsorted element 5 bubbles to second-to-last position) Compare (arr[0], arr[1]) -> (1, 4):

- 1 is NOT > 4, no swap. arr remains [1, 4, 2, 5, 8]

Compare (arr[1], arr[2]) -> (4, 2):

- 4 > 2, so swap. arr becomes [1, 2, 4, 5, 8]

Compare (arr[2], arr[3]) -> (4, 5):

- 4 is NOT > 5, no swap. arr remains [1, 2, 4, 5, 8]

End of Pass 2. The next largest element 5 is now at its correct position. The sorted part is [5, 8]. The unsorted part is [1, 2, 4].

Pass 3: (Largest remaining unsorted element 4 bubbles to third-to-last position) Compare (arr[0], arr[1]) -> (1, 2):

- 1 is NOT > 2, no swap. arr remains [1, 2, 4, 5, 8]

Compare (arr[1], arr[2]) -> (2, 4):

- 2 is NOT > 4, no swap. arr remains [1, 2, 4, 5, 8]

End of Pass 3. The next largest element 4 is now at its correct position. The sorted part is [4, 5, 8]. The unsorted part is [1, 2].

Pass 4: (Largest remaining unsorted element 2 bubbles to fourth-to-last position) Compare (arr[0], arr[1]) -> (1, 2):

- 1 is NOT > 2, no swap. arr remains [1, 2, 4, 5, 8]

End of Pass 4. The next largest element 2 is now at its correct position. The sorted part is [2, 4, 5, 8]. The unsorted part is [1].

The array is now [1, 2, 4, 5, 8], which is sorted. Although the final element 1 is implicitly sorted after Pass 4, a typical implementation might run one more loop iteration to confirm no swaps were made. In many implementations, an optimization flag can track if any swaps occurred in a pass. If no swaps, the array is sorted, and the algorithm can terminate early.

Result: [1, 2, 4, 5, 8]

31. Why is Binary Search more efficient than Linear Search for large datasets? Explain with an example.

1. **Requirement for Sorted Data:**
    - **Binary Search:** Requires the dataset to be **sorted** in ascending or descending order. This is a crucial prerequisite.
    - **Linear Search:** Does **not** require the dataset to be sorted.
2. **Search Strategy:**
    - **Binary Search:** It works by repeatedly dividing the search interval in half. It compares the target value with the middle element of the array.
        - If the target matches the middle element, the search is complete.
        - If the target is smaller than the middle element, it eliminates the right half and continues searching in the left half.
        - If the target is larger than the middle element, it eliminates the left half and continues searching in the right half. This process effectively halves the search space with each step.
    - **Linear Search:** It sequentially checks each element in the list one by one, from the beginning to the end, until the target element is found or the end of the list is reached.
3. **Time Complexity:** This is where the efficiency difference becomes most apparent.
    - **Linear Search:**
        - **Worst-case time complexity: O(n)** (Linear time). In the worst case, the algorithm might have to check every single element in the list (e.g., if the element is at the very end or not present at all).
        - For an array with n elements, it could take up to n comparisons.
    - **Binary Search:**
        - **Worst-case time complexity: O(log n)** (Logarithmic time). Each comparison effectively halves the remaining search space. The number of comparisons needed is proportional to the logarithm (base 2) of the number of elements.

- For an array with n elements, it takes at most approximately $\log_2 n$ comparisons.

The Power of Logarithmic Growth: The difference between O(n) and O(log n) is immense for large datasets.

- If you have **1000** elements:
  - Linear Search might take up to **1000** comparisons.
  - Binary Search would take approximately $\log_2 1000$ which is roughly **10** comparisons.
- If you have **1,000,000** (one million) elements:
  - Linear Search might take up to **1,000,000** comparisons.
  - Binary Search would take approximately $\log_2 1{,}000{,}000$ which is roughly **20** comparisons.

As you can see, for large datasets, Binary Search performs exponentially fewer operations than Linear Search, making it vastly more efficient.

Example: Searching for a number in a sorted array of 100 numbers

Let's say we have a sorted array of 100 numbers: [1, 2, 3, ..., 50, ..., 98, 99, 100] And we are looking for the number 95.

1. Linear Search:

- Start at 1. Is it 95? No.
- Move to 2. Is it 95? No.

  ...
- Continue checking one by one until you reach 95.
- In the worst case (if 95 was the last element, or not present), you would perform **100 comparisons**. In this specific case, you'd perform 95 comparisons to find 95.

2. Binary Search:

- **Initial search space:** [1, ..., 100] (size 100)
- **Step 1:** Find the middle element. (1 + 100) / 2 = 50. Compare 95 with 50. Since 95 > 50, discard the left half.
  - New search space: [51, ..., 100] (size 50)
- **Step 2:** Find the middle element of the new space. (51 + 100) / 2 = 75. Compare 95 with 75. Since 95 > 75, discard the left half.

- New search space: [76, ..., 100] (size 25)

- **Step 3:** Find the middle element. (76 + 100) / 2 = 88. Compare 95 with 88. Since 95 > 88, discard the left half.
  - New search space: [89, ..., 100] (size 12)
- **Step 4:** Find the middle element. (89 + 100) / 2 = 94. Compare 95 with 94. Since 95 > 94, discard the left half.
  - New search space: [95, ..., 100] (size 6)
- **Step 5:** Find the middle element. (95 + 100) / 2 = 97. Compare 95 with 97. Since 95 < 97, discard the right half.
  - New search space: [95, 96] (size 2)
- **Step 6:** Find the middle element. (95 + 96) / 2 = 95. Compare 95 with 95. Match found!

32. . Compare and contrast character arrays and strings in C. Write a C program to concatenate two strings without using the strcat() function.

A. Compare and Contrast Character Arrays and Strings in C

In C, the terms "character array" and "string" are very closely related, but they are not exactly the same. Understanding their subtle differences is crucial for effective C programming.

1. Character Array:

- **Definition:** A character array is simply an array where each element is of the char data type. Like any other array (e.g., an int array), it's a contiguous block of memory used to store a sequence of characters.
- **Purpose:** Can be used to store any sequence of characters, whether it represents text or just raw character data.
- **Termination:** A character array does **not necessarily** have a null terminator (\0) at its end. If you put characters into it, they just fill up the allocated space.
- **Example Declaration/Initialization:**

```
char myCharArray[5] = {'H', 'e', 'l', 'l', 'o'}; // No null terminator here
```

char buffer[10]; // An empty character array

- **Usage:** You can access individual characters using array indexing (e.g., myCharArray[0]). If it doesn't have a null terminator, you cannot use standard string functions (printf("%s", ...) or strlen()) on it safely, as they expect a null terminator to know where the string ends.

2. String (in C):

- **Definition:** In C, a "string" is formally defined as a **null-terminated sequence of characters**. This means it's a character array that *must* have a null character (\0) as its very last character. The null terminator signals the end of the string to C's standard library functions.
- **Purpose:** Specifically designed to store and manipulate human-readable text.
- **Termination: Always** terminated by a null character (\0). This is how functions like printf("%s", ...) know where to stop printing, or strlen() knows where to stop counting characters.
- **Example Declaration/Initialization:**

```
char greeting1[6] = {'H', 'e', 'l', 'l', 'o', '\0'}; // Explicit null terminator

char greeting2[] = "World"; // Compiler automatically adds '\0' at the end

                    // (size will be 6: 'W','o','r','l','d','\0')

char *name = "Alice"; // String literal. 'name' is a pointer to read-only memory.
```

- **Usage:** Can be safely used with standard string functions from <string.h> (like strlen(), strcpy(), strcat(), strcmp(), etc.) and printf("%s", ...) for printing.

B. C Program to Concatenate Two Strings Without strcat()

To concatenate two strings without strcat(), we need to manually copy characters from the first string to a destination array, then find the end of the first string in that destination, and finally copy the characters of the second string starting from that point.

```
#include <stdio.h> // For printf and scanf

#include <string.h> // For strlen (used for demonstration, but not for concatenation logic itself)

int main() {

  // Declare two source strings.

  // Ensure they are large enough to hold the content.

  char str1[50]; // First string, max 49 chars + '\0'

  char str2[50]; // Second string, max 49 chars + '\0'

  // Declare a destination array for the concatenated string.

  // Its size should be large enough to hold both strings plus one for the null terminator.
```

```
// 50 (max for str1) + 50 (max for str2) + 1 (for '\0') = 101
char concatenated_str[101];
int i, j; // Loop counters
// --- Input Phase ---
printf("Enter the first string: ");
// Using fgets to safely read strings, preventing buffer overflows.
// It reads up to size-1 characters or until a newline.
// The newline character is included if read.
fgets(str1, sizeof(str1), stdin);
// Remove the trailing newline character that fgets might read
str1[strcspn(str1, "\n")] = '\0';
printf("Enter the second string: ");
fgets(str2, sizeof(str2), stdin);
// Remove the trailing newline character
str2[strcspn(str2, "\n")] = '\0';
// --- Concatenation Logic (without strcat()) ---
// Step 1: Copy characters from the first string (str1) to the destination (concatenated_str)
// Loop until the null terminator of str1 is encountered
for (i = 0; str1[i] != '\0'; i++) {
    concatenated_str[i] = str1[i];
}
// After this loop, 'i' will be at the index where the null terminator of str1 was,
// which is exactly where the second string should start in concatenated_str.
// Step 2: Copy characters from the second string (str2) to the destination,
// starting from where the first string ended.
// 'j' is the index for str2, 'i' continues from its last position for concatenated_str.
```

```
    for (j = 0; str2[j] != '\0'; j++, i++) {
        concatenated_str[i] = str2[j];
    }
    // Step 3: Add the null terminator to the end of the concatenated string.
    // This is crucial to make it a valid C string.
    concatenated_str[i] = '\0';
    // --- Output Phase ---
    printf("\nOriginal String 1: \"%s\"\n", str1);
    printf("Original String 2: \"%s\"\n", str2);
    printf("Concatenated String: \"%s\"\n", concatenated_str);
    // Optional: Demonstrate strlen to confirm length
    printf("Length of concatenated string: %zu\n", strlen(concatenated_str));
    return 0;
}
```

Explanation of Concatenation Logic:

1. **Initialization:**
    - str1 and str2 are declared to hold the input strings.
    - concatenated_str is declared with a size large enough to hold the combined length of str1 and str2, plus one extra byte for the null terminator (\0). This is crucial to prevent buffer overflows.
    - i and j are loop counters.
2. **Input Reading (fgets and strcspn):**
    - fgets(str1, sizeof(str1), stdin); is used for safer string input than scanf("%s", ...). fgets reads characters from stdin (standard input) into str1 until sizeof(str1) - 1 characters are read, a newline character \n is encountered, or the end-of-file is reached.

- A common issue with fgets is that it includes the newline character \n if the user presses Enter before the buffer is full. str1[strcspn(str1, "\n")] = '\0'; is a standard idiom to remove this trailing newline by replacing it with a null terminator.

3. **Copying the First String (str1):**
   - for (i = 0; str1[i] != '\0'; i++) { concatenated_str[i] = str1[i]; }
   - This for loop iterates through str1 character by character, starting from index 0.
   - It continues as long as the current character str1[i] is not the null terminator (\0).
   - In each iteration, str1[i] is copied to concatenated_str[i].
   - When the loop finishes, i will hold the index *immediately after* the last character copied from str1. This is precisely where str2 needs to begin.
4. **Copying the Second String (str2):**
   - for (j = 0; str2[j] != '\0'; j++, i++) { concatenated_str[i] = str2[j]; }
   - This loop copies characters from str2.
   - j is used as the index for str2, starting from 0.
   - i (which retained its value from the previous loop) is used as the index for concatenated_str. Both j and i are incremented in each step.
   - Characters from str2 are copied to concatenated_str starting from the position where str1 ended.
5. **Null Termination:**
   - concatenated_str[i] = '\0';
   - After both loops complete, i will be at the position where the last character of str2 was copied, plus one. This is the correct spot to place the null terminator. Adding \0 makes concatenated_str a valid C string, allowing printf("%s", ...) and strlen() to work correctly.

33. Differentiate between call by value and call by reference in functions. Write a program to demonstrate the difference.

Differentiating Call by Value and Call by Reference

1. Call by Value

- **Mechanism:** When you pass arguments by value, a **copy** of the actual argument's value is made and passed to the function's formal parameter.

- **Memory:** The formal parameter (inside the function) gets its own separate memory location, distinct from the actual argument (in the calling function, e.g., main).

- **Modification:** Any changes made to the formal parameter inside the function **do not affect** the original actual argument in the calling function. The function works on a copy.

- **Use Case:** Use call by value when you want the function to operate on the data without altering the original variable in the calling scope. It's suitable for passing simple data types like integers, floats, characters, etc., when you only need to read their values.

2. Call by Reference (using Pointers in C)

- **Mechanism:** When you pass arguments by reference (in C, this is achieved by passing **pointers** to the variables), the **memory address** of the actual argument is passed to the function's formal parameter.

- **Memory:** The formal parameter (which is a pointer) stores the address of the original actual argument. Both the formal parameter and the actual argument refer to the *same* memory location.

- **Modification:** Any changes made to the data *at the memory address pointed to* by the formal parameter **will affect** the original actual argument in the calling function. The function is working directly on the original data.

- **Use Case:** Use call by reference when you need a function to modify the original variables in the calling scope. It's essential for tasks like swapping values, modifying multiple variables, or when passing large data structures (like arrays or complex structs) to avoid copying overhead.

Program to Demonstrate the Difference

This program will attempt to "swap" two numbers using both call by value and call by reference, clearly showing why call by reference is necessary for modifying original variables.

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

// --- Function demonstrating Call by Value ---

// This function tries to swap two integers passed by value.

// It will NOT affect the original variables in main.

void swapByValue(int a, int b) {

  printf("\n--- Inside swapByValue function ---\n");

  printf("Before swap: a = %d, b = %d\n", a, b);

  int temp = a; // Store the original value of 'a'

  a = b;      // 'a' now gets the value of 'b'
```

```
    b = temp;     // 'b' now gets the original value of 'a'
    printf("After swap:  a = %d, b = %d\n", a, b);
    printf("---------------------------------\n");}
// --- Function demonstrating Call by Reference ---
// This function correctly swaps two integers passed by reference (using pointers).
// It WILL affect the original variables in main.
void swapByReference(int *ptrA, int *ptrB) {
    printf("\n--- Inside swapByReference function ---\n");
    // Dereference pointers to access and modify the values at their addresses
    printf("Before swap: *ptrA = %d, *ptrB = %d\n", *ptrA, *ptrB);
    int temp = *ptrA; // Store the value pointed to by ptrA
    *ptrA = *ptrB;    // Value pointed to by ptrA gets value pointed to by ptrB
    *ptrB = temp;     // Value pointed to by ptrB gets original value pointed to by ptrA
    printf("After swap:  *ptrA = %d, *ptrB = %d\n", *ptrA, *ptrB);
    printf("---------------------------------\n"); }
int main() {
    int num1 = 10;
    int num2 = 20;
    printf("--- Main function before any swaps ---\n");
    printf("Initial values: num1 = %d, num2 = %d\n", num1, num2);
    // --- Demonstrate Call by Value ---
    printf("\nCalling swapByValue(num1, num2)...\n");
    swapByValue(num1, num2); // Passing copies of num1 and num2
    printf("\n--- Main function after swapByValue ---\n");
    // Observe that num1 and num2 are UNCHANGED here
    printf("Values in main: num1 = %d, num2 = %d\n", num1, num2);
```

```
    printf("-----------------------------------\n");

    // --- Demonstrate Call by Reference ---

    printf("\nCalling swapByReference(&num1, &num2)...\n");

    // Passing the ADDRESSES of num1 and num2 using the '&' operator

    swapByReference(&num1, &num2);

    printf("\n--- Main function after swapByReference ---\n");

    // Observe that num1 and num2 ARE CHANGED here

    printf("Values in main: num1 = %d, num2 = %d\n", num1, num2);

    printf("-----------------------------------\n");

    return 0; // Indicate successful program execution

}
```

34. Write a recursive function to find the Fibonacci series up to n terms and explain how recursion works internally.

The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. The sequence goes: 0,1,1,2,3,5,8,13,21,...

Mathematically, it's defined as:

- F(0)=0
- F(1)=1
- F(n)=F(n−1)+F(n−2) for n>1

C Program: Fibonacci Series using Recursion

```
#include <stdio.h> // Standard input/output library

// Recursive function to calculate the n-th Fibonacci number

// F(n) = F(n-1) + F(n-2)

long long fibonacci(int n) {

    // Base Case 1: If n is 0, the 0th Fibonacci number is 0.
```

```c
    if (n == 0) {
        return 0;
    }
    // Base Case 2: If n is 1, the 1st Fibonacci number is 1.
    else if (n == 1) {
        return 1;
    }
    // Recursive Step: For n > 1, sum the (n-1)th and (n-2)th Fibonacci numbers.
    else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
int main() {
    int num_terms; // To store the number of terms the user wants
    int i;       // Loop counter
    printf("Enter the number of terms for the Fibonacci series: ");
    scanf("%d", &num_terms);
    // Input Validation
    if (num_terms < 0) {
        printf("Error: Please enter a non-negative number of terms.\n");
        return 1; // Indicate an error
    } else if (num_terms == 0) {
        printf("Fibonacci Series: (No terms requested)\n");
    } else {
        printf("Fibonacci Series up to %d terms:\n", num_terms);
        // Loop from 0 to num_terms-1 to print each Fibonacci number
```

```
    for (i = 0; i < num_terms; i++) {
        // Call the recursive fibonacci function for each term
        printf("%lld ", fibonacci(i));
    }
    printf("\n");
  }
  return 0; // Indicate successful execution
}
```

How Recursion Works Internally (The Call Stack)

When a function calls itself (or any other function), the computer uses a data structure called the **Call Stack** (or just "stack") to manage the execution flow. Each time a function is called, a new **stack frame** (also known as an activation record) is pushed onto the top of the stack. This stack frame contains:

- The function's local variables.
- The function's parameters.
- The return address (where the program should resume execution after the function finishes).
- Other administrative information.

When a function finishes executing (either by reaching a return statement or the end of its code), its stack frame is popped off the stack, and control returns to the address specified in the return address of the previous stack frame.

Let's trace the execution of fibonacci(4) to understand this internal process:

Goal: Calculate fibonacci(4)

1. **main() calls fibonacci(4)**
    - A stack frame for fibonacci(4) is pushed.
    - Inside fibonacci(4): n is 4.
    - if (4 == 0) and else if (4 == 1) are false.
    - It reaches return fibonacci(3) + fibonacci(2);

- fibonacci(3) is called first.

2. **fibonacci(4) calls fibonacci(3)**
    - A new stack frame for fibonacci(3) is pushed on top of fibonacci(4)'s frame.
    - Inside fibonacci(3): n is 3.
    - if (3 == 0) and else if (3 == 1) are false.
    - It reaches return fibonacci(2) + fibonacci(1);
    - fibonacci(2) is called first.
3. **fibonacci(3) calls fibonacci(2)**
    - A new stack frame for fibonacci(2) is pushed.
    - Inside fibonacci(2): n is 2.
    - if (2 == 0) and else if (2 == 1) are false.
    - It reaches return fibonacci(1) + fibonacci(0);
    - fibonacci(1) is called first.
4. **fibonacci(2) calls fibonacci(1) (Base Case)**
    - A new stack frame for fibonacci(1) is pushed.
    - Inside fibonacci(1): n is 1.
    - else if (1 == 1) is true.
    - It return 1;
    - The stack frame for fibonacci(1) is popped. Execution returns to fibonacci(2).
5. **fibonacci(2) calls fibonacci(0) (Base Case)**
    - fibonacci(2) now needs to calculate fibonacci(0).
    - A new stack frame for fibonacci(0) is pushed.
    - Inside fibonacci(0): n is 0.
    - if (0 == 0) is true.
    - It return 0;
    - The stack frame for fibonacci(0) is popped. Execution returns to fibonacci(2).

6. **fibonacci(2) completes its calculation**
    - fibonacci(2) now has the results: fibonacci(1) returned 1, and fibonacci(0) returned 0.
    - It calculates 1 + 0 = 1.
    - It return 1;
    - The stack frame for fibonacci(2) is popped. Execution returns to fibonacci(3).
7. **fibonacci(3) calls fibonacci(1) (Base Case - again!)**
    - fibonacci(3) now needs to calculate fibonacci(1) (the right side of its + operation).
    - A new stack frame for fibonacci(1) is pushed.
    - Inside fibonacci(1): n is 1.
    - else if (1 == 1) is true.
    - It return 1;
    - The stack frame for fibonacci(1) is popped. Execution returns to fibonacci(3).
8. **fibonacci(3) completes its calculation**
    - fibonacci(3) now has the results: fibonacci(2) returned 1, and fibonacci(1) returned 1.
    - It calculates 1 + 1 = 2.
    - It return 2;
    - The stack frame for fibonacci(3) is popped. Execution returns to fibonacci(4).
9. **fibonacci(4) calls fibonacci(2) (Base Case - again!)**
    - fibonacci(4) now needs to calculate fibonacci(2) (the right side of its + operation).
    - This will again trigger fibonacci(1) and fibonacci(0) calls, similar to steps 3-6, eventually returning 1.
    - (Skipping the detailed sub-steps for brevity, as they mirror steps 3-6)
    - fibonacci(2) returns 1.
    - The stack frame for fibonacci(2) is popped. Execution returns to fibonacci(4).
10. **fibonacci(4) completes its calculation**
    - fibonacci(4) now has the results: fibonacci(3) returned 2, and fibonacci(2) returned 1.

- It calculates 2 + 1 = 3.
- It return 3;
- The stack frame for fibonacci(4) is popped. Execution returns to main().

Final Result in main(): fibonacci(4) returns 3.

35A. What are Built-in Library Functions in C?

Built-in library functions (more accurately called **Standard Library Functions**) in C are pre-written functions that are part of the C Standard Library. These functions are readily available for use by programmers without needing to write their own implementation for common tasks. They are typically optimized for performance and reliability, saving developers time and effort.

To use these functions, you need to include the appropriate header file (e.g., <stdio.h>, <string.h>, <math.h>) at the beginning of your C program using the #include preprocessor directive. This tells the compiler where to find the declarations (prototypes) of these functions.

Here are examples from at least three different categories:

1. **Input/Output Functions (from <stdio.h>)**
   - These functions handle reading data from input devices (like the keyboard) and writing data to output devices (like the screen or files).
   - **Examples:**
     - printf(): Used to print formatted output to the console.
     - scanf(): Used to read formatted input from the console.
     - fgets(): Used to read a line of text from a stream (e.g., stdin) into a string, with buffer overflow protection.
     - getchar(): Reads a single character from the standard input.
2. **String Manipulation Functions (from <string.h>)**
   - These functions are designed for common operations on C strings (null-terminated character arrays).
   - **Examples:**
     - strlen(): Calculates the length of a string (number of characters before the null terminator).
     - strcpy(): Copies one string to another.
     - strcat(): Concatenates (joins) two strings.

        - strcmp(): Compares two strings lexicographically.

3. **Mathematical Functions (from <math.h>)**
    - These functions perform common mathematical operations.
    - **Examples:**
        - sqrt(): Calculates the square root of a number.
        - pow(): Calculates a base raised to an exponent (xy).
        - sin(), cos(), tan(): Trigonometric functions.
        - fabs(): Returns the absolute value of a floating-point number.

35B. How Arrays are Passed to Functions in C

In C, arrays are **always passed by reference** (or more accurately, "call by address" or "pass by pointer") to functions. This means that when you pass an array to a function, the entire array is *not* copied. Instead, what is actually passed is:

- The **base address** (memory address of the first element) of the array.
- The **type** of the array's elements.

Key Implications:

1. **No Copy:** The function does not receive a separate copy of the array. It receives a pointer to the original array in the calling function's memory.
2. **Modifications Affect Original:** Any changes made to the elements of the array inside the function *will directly affect* the original array in the calling function (e.g., main).
3. **Size Not Implicitly Passed:** When an array is passed, its size is *not* automatically known to the function. You typically need to pass the size of the array as a separate integer argument to the function.
4. **Function Parameter Declaration:** In the function's parameter list, you can declare an array parameter in a few equivalent ways:
    - void func(int arr[], int size) (most common and readable)
    - void func(int *arr, int size) (explicitly as a pointer)
    - void func(int arr[100], int size) (specifying size, but compiler ignores it and treats it as int *)

Example: Passing an Array to a Function

This program demonstrates how an array is passed to a function, and how modifications inside the function affect the original array.

```
#include <stdio.h> // Standard input/output library

// Function to print the elements of an array

void printArray(int arr[], int size) {

   printf("Array elements: ");

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

      printf("%d ", arr[i]);

   }

   printf("\n");

}

// Function to modify an element in the array

// Since arrays are passed by reference, changes here will affect the original array

void modifyArray(int arr[], int size) {

   printf("\n--- Inside modifyArray function ---\n");

   printf("Attempting to change the first element...\n");

   // Modify the first element of the array

   arr[0] = 999; // This changes the original array's first element

   printf("Array after modification inside function: ");

   printArray(arr, size); // Print the modified array from within the function

   printf("-----------------------------------\n");

}

int main() {

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

   int arraySize = sizeof(myNumbers) / sizeof(myNumbers[0]); // Calculate array size

   printf("--- Main function ---\n");
```

```
    printf("Original array in main: ");
    printArray(myNumbers, arraySize); // Print original array
    // Call the function to modify the array
    // We pass the array name (which decays to its base address) and its size
    modifyArray(myNumbers, arraySize);
    printf("\n--- Main function after modifyArray call ---\n");
    // Observe that the original array in main has been modified
    printf("Array in main after function call: ");
    printArray(myNumbers, arraySize); // Print array again to show changes
    return 0;
}
```

Explanation of the Example:

1. **main function:**
    - An integer array myNumbers is declared and initialized.
    - arraySize is calculated to determine the number of elements.
    - printArray is called to show the initial state of myNumbers.
2. **modifyArray function:**
    - It takes int arr[] and int size as parameters. When modifyArray(myNumbers, arraySize) is called from main, arr inside modifyArray becomes a pointer to the first element of myNumbers.
    - arr[0] = 999; directly modifies the content at the memory location of myNumbers[0] in the main function.
    - The printArray call inside modifyArray shows the array *after* this modification.
3. **Output in main after modifyArray call:**
    - When control returns to main, the final printArray(myNumbers, arraySize) will show that myNumbers[0] is now 999, confirming that the modification made within modifyArray persisted in the original array.