Programming with C

UNIT – I: Computer Fundamentals , Algorithm and Basics of C

Important Questions:

1. Define computer. Explain the classification of computers.
2. What is an operating system? Describe its functions.
3. Explain the Block diagram of computer/Operational overview.
4. What are programming languages? Explain generations and classification.
5. Distinguish between compiling, interpreting, loading, and linking.
6. What are the steps involved in developing a program?
7. **Define an algorithm. What are the characteristics of a good algorithm?**
8. **Explain Structure of a c program with an example**
9. **What are tokens in C? Explain keywords, identifiers, constants, and operators.**
10. **Explain data types in C with examples.**
11. **What is type conversion? Differentiate implicit and explicit casting.**
12. **Explain briefly about expression evaluation and its precedence.**

✅UNIT – II: Input/Output, Control Statements, Arrays & Strings

Important Questions:

1. Differentiate between formatted and non-formatted I/O functions.
2. Explain the use of `printf()` and `scanf()` with examples.
3. What are escape sequences? List any 5 with meanings.
4. Write a C program to check if a number is prime.
5. Write a C program to find the sum and average of an array.
6. Write a program to reverse a string.
7. **Explain selection control statements: `if`, `if-else`, `nested if-else`, `switch`.**
8. **Explain `while`, `do-while`, and `for` loops with flowcharts and examples.**
9. **Differentiate between `break`, `continue`, and `goto` statements.**
10. **What is an array? Explain about types of arrays with examples?**
11. **Explain character arrays and string functions (`strlen`, `strcpy`, `strcmp`, `strcat`) from `string.h`.**

✅UNIT – III: Functions & Pointers

Important Questions:

1. Explain about different types of functions
2. How to pass arrays to functions? Write an example.
3. Write a program to swap two numbers using pointers.
4. Explain pointer to pointer and pointer to array with diagrams.
5. **Explain call-by-value and call-by-reference with examples.**
6. **Write a recursive function to calculate factorial.**
7. **What is the scope and lifetime of a variable in C?**
8. **Explain the different storage classes in C.**
9. **What is a pointer? Explain pointer declaration and initialization.**
10. **What is a function? Explain different parts of function.**
11. **What is dynamic memory allocation? Explain `malloc()`, `calloc()`, and `free()`.**

✅UNIT – IV: Structures, Unions, and File Handling

Important Questions:

1. Write a program using structures to store student data (roll, name, marks).
2. How to access members of a structure using a pointer?
3. What is a union? How is it different from a structure?
4. Write a C program to copy content from one file to another.
5. How to use `fgets()` and `fputs()` for file input and output?
6. Explain file access modes (`r`, `w`, `a`, `r+`, etc.).
7. **What is a structure in C? How do you declare and use it?**
8. **Compare structure and union with examples.**
9. **What is an enumerated type? Give an example of its usage.**
10. **Explain file handling functions in C: `fopen()`, `fclose()`, `fprintf()`, `fscanf()`.**
11. **Differentiate between text and binary files.**

Programming with C

UNIT – I: Computer Fundamentals , Algorithm and Basics of C

Q 1. Define computer. Explain the classification of computers.

1. Definition of Computer

A **computer** is an electronic device that accepts data as input, processes it using a set of instructions (programs), and produces meaningful results (output). It can store data, retrieve it, and perform both simple and complex calculations at very high speed and accuracy.

In simple terms:
A computer is a programmable machine that performs arithmetic and logical operations automatically.

---

2. Classification of Computers

Computers can be classified in several ways. The main classifications are:

A. Based on Operating Principle

1. Analog Computers

- Work with continuous data.
- Measure physical quantities like temperature, speed, pressure, etc.
- Example: Speedometer, analog voltmeter.

2. Digital Computers

- Work with discrete numbers (0s and 1s).
- Most commonly used today.
- Example: PCs, laptops, calculators.

3. Hybrid Computers

- Combine features of both analog and digital computers.
- Used in specialized applications like hospitals and scientific research.

B. Based on Size and Processing Power

1. Microcomputers

- Smallest in size.
- Used by individuals.
- Examples: Desktop computers, laptops, tablets, smartphones.

2. Minicomputers

- Medium-sized.
- Support multiple users simultaneously.
- Used in small businesses and laboratories.

3. Mainframe Computers

- Very powerful.
- Handle large amounts of data.
- Support hundreds or thousands of users at once.
- Used in banks, railways, large organizations.

4. Supercomputers

- Most powerful and fastest computers.
- Used for complex scientific calculations, weather forecasting, space research.

C. Based on Purpose

1. General-Purpose Computers

- Designed to perform a wide variety of tasks.
- Examples: PCs, laptops.

2. Special-Purpose Computers

- Designed for a specific task.
- Examples: ATM machines, washing machine controllers, traffic light systems.

Q 2. What is an Operating System?

An **Operating System (OS)** is system software that acts as an interface between the **user** and the **computer hardware**.
It manages hardware resources, runs applications, and provides a user-friendly environment to operate the computer.

In simple terms:
An operating system controls the working of computer hardware and software, making it possible for users to perform tasks smoothly.

Examples: Windows, Linux, macOS, Android, iOS.

Functions of an Operating System

The major functions of an operating system include:

---

1. Process Management

- Manages creation, scheduling, and termination of processes (programs in execution).
- Allocates CPU time to different processes.
- Ensures multitasking.

---

2. Memory Management

- Keeps track of each memory location.
- Allocates and deallocates memory space to programs.
- Ensures optimal use of RAM.

---

3. File Management

- Manages files on storage devices.
- Handles operations like creating, opening, reading, writing, and deleting files.
- Maintains directory structure.

---

4. Device Management

- Controls and coordinates the use of hardware devices (printers, keyboards, disks, etc.).
- Uses device drivers to communicate with hardware.

---

5. Input/Output (I/O) Management

- Manages input and output operations efficiently.
- Ensures smooth data transfer between I/O devices and system memory.

---

6. Security and Protection

- Protects data and system resources from unauthorized access.
- Provides user authentication (passwords, permissions, encryption).

---

7. User Interface Management

- Provides interaction methods such as:
  - **GUI (Graphical User Interface)** – Windows, icons, menus.
  - **CLI (Command Line Interface)** – Commands typed by the user.

---

8. Error Detection and Handling

- Detects system errors and takes corrective actions.
- Provides warnings and diagnostic messages.

9. Resource Allocation

- Allocates hardware resources like CPU, memory, storage, and I/O devices to multiple users or tasks efficiently.

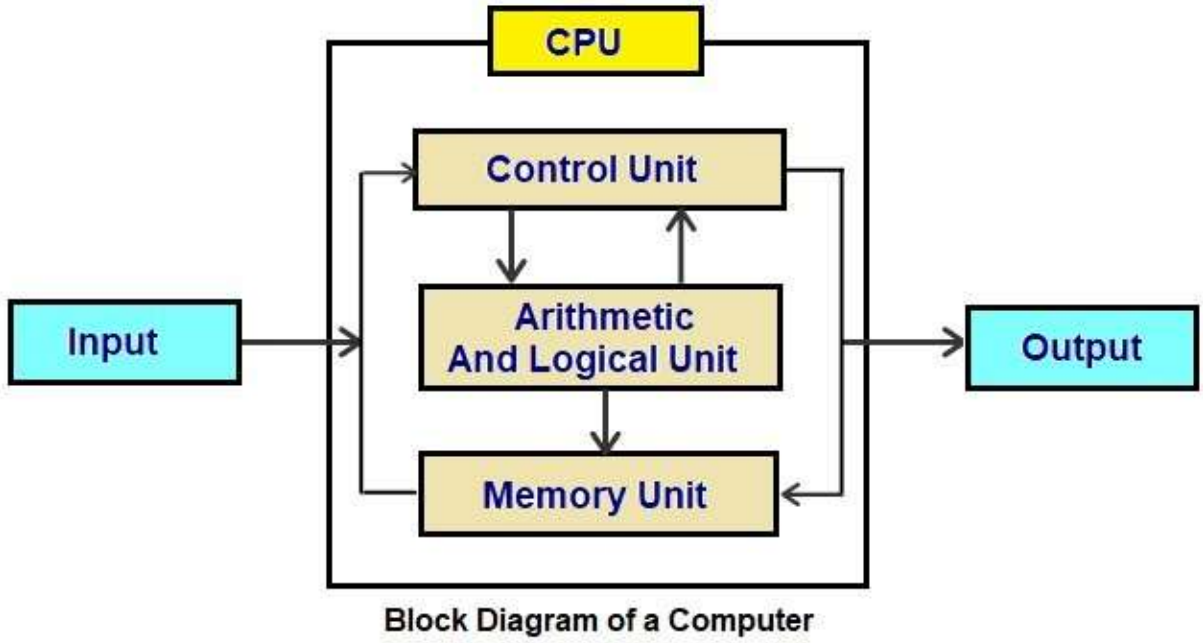
10. Accounting and Monitoring

- Keeps logs of system usage.
- Helps in performance monitoring and optimizing the system.

Q 3. Explain the Block diagram of computer/Operational overview.

Block Diagram of a Computer / Operational Overview

A **computer system** works by performing four basic operations:
Input → Processing → Output → Storage
The block diagram shows how different units of a computer work together to perform tasks.



Block Diagram of a Computer

Main Components of the Block Diagram

✓ 1. Input Unit

- Accepts data and instructions from the user.
- Converts them into machine-understandable form.
- Sends data to the CPU for processing.

Examples: Keyboard, Mouse, Scanner, Microphone.

✓ 2. Central Processing Unit (CPU)

The **CPU** is known as the "brain of the computer."
It performs all processing activities.

It has three main parts:

a) Arithmetic and Logic Unit (ALU)

- Performs arithmetic operations: addition, subtraction, multiplication, division.
- Performs logical operations: comparisons (>, <, =).

b) Control Unit (CU)

- Controls and coordinates all operations of the computer.
- Tells input, output, and memory what to do.
- Fetches instructions from memory and executes them.

c) Registers

- Small, high-speed storage locations inside the CPU.
- Temporarily hold data and instructions during processing.

✓ 3. Memory Unit

Memory stores data temporarily or permanently.

It is divided into:

a) Primary Memory (Main Memory/RAM & ROM)

- Stores data and programs currently in use.
- Fast but limited in size.
- **RAM** → Temporary storage, volatile.
- **ROM** → Permanent instructions, non-volatile.

b) Secondary Memory

- Stores data permanently.
- Larger in capacity but slower.

Examples: Hard disk, SSD, CD/DVD, Pen Drive.

✓ 4. Output Unit

- Presents the processed information (results) to the user.
- Converts machine information into human-readable form.

Examples: Monitor, Printer, Speakers.

Operational Overview (How a Computer Works)

1. **Input Unit** receives data and instructions.
2. Data is sent to **Memory** where it is stored temporarily.
3. The **Control Unit (CU)** fetches instructions from memory.
4. **ALU** performs calculations and logical decisions.
5. Results are stored back in **Memory**.
6. Results are sent to the **Output Unit** for display.

This cycle is known as the "Input–Process–Output (IPO)" cycle.

Q 4. What are Programming Languages?

A **programming language** is a set of instructions, commands, and syntax used to write programs that a computer can understand and execute.
It allows programmers to communicate with the computer and develop software, applications, and systems.

Examples: C, C++, Java, Python, HTML, JavaScript, etc.

Generations of Programming Languages

Programming languages have evolved through **five generations**, each representing a higher level of abstraction and ease of use.

---

1. First Generation Languages (1GL) – Machine Language

- Written in **binary (0s and 1s)**.
- Directly understood by the computer.
- Very fast but difficult to write and debug.
- Machine-dependent.

Example:

```
10110011 00001111
```

2. Second Generation Languages (2GL) – Assembly Language

- Uses **mnemonics** instead of binary.
- Easier than machine language but still hardware-dependent.
- Requires an **assembler** to convert to machine code.

Examples:

```
MOV A, B, ADD X
```

---

3. Third Generation Languages (3GL) – High-Level Languages

- Use English-like statements.
- Programmer-friendly and machine-independent.
- Require a **compiler or interpreter**.
- Widely used for general application development.

Examples: C, C++, Java, Python, FORTRAN, COBOL.

4. Fourth Generation Languages (4GL) – Very High-Level Languages

- Designed to increase productivity.
- Mostly non-procedural (focus on what to do, not how).
- Used in database queries, report generation, and automation.

  Examples: SQL, MATLAB, Oracle Reports, SAS.

---

5. Fifth Generation Languages (5GL) – Artificial Intelligence Languages

- Used in **AI**, **machine learning**, **neural networks**, and **expert systems**.
- Rely on logic and constraints rather than traditional coding.

Examples: Prolog, LISP, Mercury.

---

5. Distinguish Between Compiling, Interpreting, Loading, and Linking

The four terms are related to how a program is translated and executed.
Here is the distinction among them:

---

1. Compiling

- Converts the entire high-level program into machine code at once.
- Produces an **object file** or **executable file**.
- Errors are shown **after** the whole program is scanned.
- Execution is **faster** after successful compilation.

Tools used: Compiler
Examples: C, C++, Java (partly)

---

2. Interpreting

- Translates and executes the program **line by line**.
- Stops immediately when an error occurs.
- Does **not** generate an object file.
- Execution is **slower** compared to compiled programs.

Tools used: Interpreter
Examples: Python, JavaScript, Ruby

---

3. Loading

- The process of placing the compiled program (machine code) from secondary memory into **main memory (RAM)** for execution.
- Done by the **loader** (part of the operating system).
- Must happen before a program starts running.

---

4. Linking

- Combines the main program with other required files such as:

    - library functions,
    - header files,
    - modules,
    - external code.
- Produces a single complete executable file.
- Done by a **linker**.

Table: Difference at a Glance

Concept	What it Does	How it Works	Output	Tool Used
Compiling	Translates entire program to machine code	All at once	Object file / executable	Compiler
Interpreting	Executes program line by line	One line at a time	No object file	Interpreter
Loading	Moves program into RAM for execution	Before execution	Program in memory	Loader
Linking	Combines program with libraries	After compilation	Final executable file	Linker

---

6. What Are the Steps Involved in Developing a Program?

Developing a program follows a systematic process called the **Program Development Life Cycle (PDLC)**.
It ensures that the program is correct, efficient, and easy to maintain.

Here are the main steps:

---

1. Problem Definition

- Clearly understand and define the problem.
- Identify inputs, expected outputs, and constraints.
- Helps avoid misunderstanding and errors.

---

2. Feasibility Study / Planning

- Decide whether the problem can be solved with programming.
- Determine required time, cost, and resources.

---

3. Designing the Solution

- Create a plan or blueprint for solving the problem.
- Use tools such as:
  - **Algorithms**
  - **Flowcharts**
  - **Pseudocode**
- Describes the logic of the program.

---

4. Coding (Writing the Program)

- Convert the design into a program using a programming language (C, Python, Java, etc.).
- Follow proper syntax and programming style.

---

5. Testing and Debugging

- Run the program to check for errors (bugs).
- Correct syntax errors, logical errors, and runtime errors.
- Ensure the program works as expected under all conditions.

---

6. Documentation

- Prepare user manuals and technical documentation.
- Helps future maintenance, updates, and understanding of the program.

---

7. Execution and Implementation

- Install and run the program in a real working environment.
- Provide training to users if needed.

---

8. Maintenance

- Update, modify, or improve the program after implementation.
- Fix new bugs, add new features, and ensure smooth functioning.

Short Summary (for fast revision)

1. Problem Definition
2. Planning/Feasibility
3. Designing (Algorithm/Flowchart)
4. Coding
5. Testing & Debugging
6. Documentation
7. Implementation
8. Maintenance

---

7. Define an Algorithm. What are the Characteristics of a Good Algorithm?

Definition of Algorithm

An **algorithm** is a step-by-step procedure or a set of well-defined instructions used to solve a problem or complete a task.
It takes some input, processes it, and produces an output.

In simple terms:
An algorithm is a sequence of logical steps to achieve a specific goal.

Characteristics of a Good Algorithm

A good algorithm should have the following characteristics:

---

1. Input

- An algorithm should have **zero or more** well-defined inputs.

---

2. Output

- Must produce at least **one output** that clearly solves the problem.

---

3. Definiteness

- Each step must be clear, unambiguous, and precise.
- No vague instructions.

4. Finiteness

- The algorithm must always terminate after a **finite number of steps**.

5. Effectiveness

- Each step must be basic, simple, and feasible to perform with available resources.

6. Correctness

- The algorithm must produce the **correct output** for all valid inputs.

7. Generality

- Should be applicable to a **set of input values**, not just one specific case.

8. Efficiency (Optional but important)

- Should use minimum time and memory resources.

---

8. Explain the Structure of a C Program with an Example

A C program follows a specific structure consisting of several sections.
Below is the general structure followed by an example.

Structure of a C Program

A standard C program has the following parts:

---

1. Documentation Section (Comments)

- Used to write information about the program.
- Helps understand program purpose.

Example:

```
/* Program to add two numbers */
```

---

2. Link/Preprocessor Section

- Includes header files that contain library functions.

Example:

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

---

3. Definition Section

- Used to define constants or macros.

Example:

```
#define PI 3.14
```

---

4. Global Declaration Section

- Declare global variables and user-defined functions.

Example:

```
int x, y;
```

5. main() Function Section

This is the **starting point** of every C program.
It contains:

a) Declaration Part

Declares variables inside main.

b) Executable Part

Contains statements that perform operations.

Example:

```
int a, b, sum;   // declaration
sum = a + b;     // executable
```

6. Subprograms/Function Definition Section

- User-defined functions are written here.
- These functions are called from main().

Diagrammatic View

```
--------------------------------------
| Documentation Section              |
--------------------------------------
| Link Section (#include)            |
--------------------------------------
| Definition Section (#define)       |
--------------------------------------
| Global Declarations                |
--------------------------------------
| main() Function                    |
|   - Local Declarations             |
|   - Executable Statements          |
--------------------------------------
| User-defined Functions             |
--------------------------------------
```

Example C Program

```
/* Program to add two numbers */

#include <stdio.h>

int main()      // main function
{
    int a, b, sum;    // declaration section

    a = 5;            // executable statements
    b = 10;
    sum = a + b;

    printf("Sum = %d", sum);  // output result

    return 0;        // program ends
}
```

Explanation of Example

- **Comments:** Describe the program.
- **#include <stdio.h>:** Includes standard I/O library.
- **main():** Starting point of the program.
- **Variables (a, b, sum):** Declared inside main.
- **Executable statements:** Assign values and compute sum.
- **printf():** Displays result.
- **return 0:** Ends the program.

---

9. What Are Tokens in C?

In C programming, **tokens** are the smallest individual units or building blocks of a program. Just like words form sentences in English, tokens form C statements.

Examples of tokens:

- Keywords
- Identifiers
- Constants
- Operators
- Strings
- Punctuators (symbols like `;`, `{}`, `()`)

In this question, we explain **keywords, identifiers, constants, and operators**.

---

1. Keywords

- Keywords are **predefined reserved words** in C.
- They have a **fixed meaning** and cannot be used as variable names.
- All keywords are **written in lowercase**.

Examples (C has 32 keywords):
`int`, `float`, `if`, `else`, `for`, `while`, `return`, `char`, `void`, `switch`.

Example use:

```
int number;
```

---

2. Identifiers

- Identifiers are **names given by the programmer** to variables, functions, arrays, etc.
- They are user-defined.

Identifier rules:

- Must start with a **letter or underscore**.
- Cannot start with a digit.
- No spaces and no special characters except _.
- Case-sensitive.
- Cannot be a keyword.

Examples:
`age`, `totalMarks`, `_sum`, `student1`.

Valid: `price_1`
Invalid: `2value`, `my-name`, `float (keyword)`

---

3. Constants

- Constants are **fixed values** that do not change during program execution.
- They can be of different types:

Types of constants:

a) Integer constants

Example: `10`, `-45`, `0`

b) Floating-point constants

Example: `3.14`, `-0.99`

c) Character constants

Single character inside single quotes
Example: `'A'`, `'9'`, `'#'`

d) String constants

Characters inside double quotes
Example: `"Hello"`, `"C Programming"`

e) Symbolic constants (using #define)

Example:

```
#define PI 3.14
```

---

4. Operators

Operators are **symbols** used to perform operations on variables and values.

Types of operators in C:

a) Arithmetic Operators

`+`, `-`, `*`, `/`, `%`

b) Relational Operators

`>`, `<`, `>=`, `<=`, `==`, `!=`

c) Logical Operators

`&&`, `||`, `!`

d) Assignment Operators

`=, +=, -=, *=, /=`

e) Increment/Decrement

`++, --`

f) Bitwise Operators

`&, |, ^, <<, >>`

g) Ternary Operator

`? :`

h) Special Operators

`sizeof`, comma `,`, pointer operator `*`, address operator `&`

Short

Tokens: **Smallest units of a C program.**

- **Keywords:** Reserved words with fixed meaning.
- **Identifiers:** User-defined names for variables/functions.
- **Constants:** Fixed values (cannot change).
- **Operators:** Symbols used to perform operations.

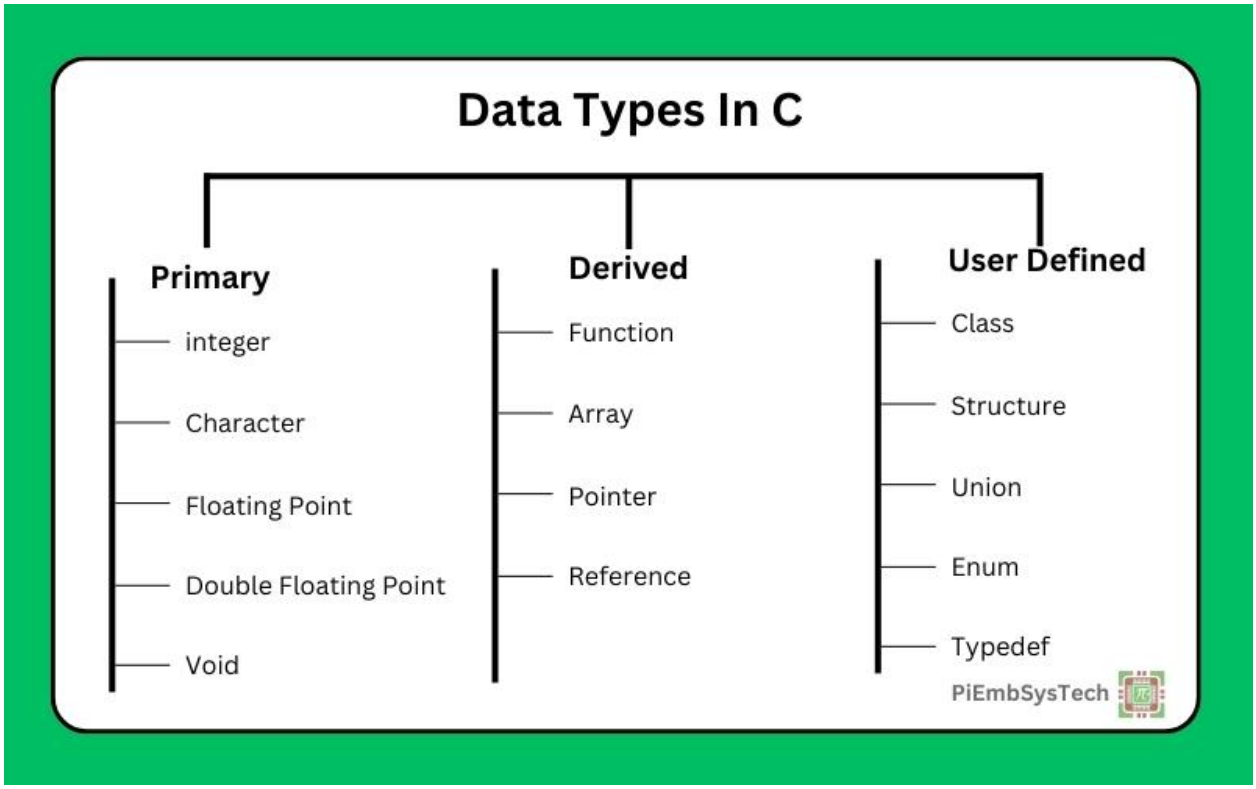
---

9. Explain data types in C with examples.

Data Types in C (Detailed Explanation with Examples)

In C, **data types define the type of data a variable can store**, such as integers, characters, decimals, or user-defined structures. They also determine the **memory size**, **range**, and **operations** allowed on the variable.

C categorizes data types into the following groups:



1 Primary (Basic / Built-in) Data Types

These are the fundamental data types provided by C.

a. int

- Stores integer values (whole numbers, positive or negative)
- Typical size: **4 bytes**
- Range (32-bit): **–2,147,483,648 to 2,147,483,647**

Example:

```
int age = 20;
```

---

b. float

- Stores **single-precision** floating-point numbers
- Size: **4 bytes**
- Precision: **6–7 decimal places**

Example:

```
float price = 99.75;
```

---

c. double

- Stores **double-precision** floating-point numbers
- Size: **8 bytes**
- Precision: **15 decimal places**

Example:

```
double distance = 12345.6789123;
```

---

d. char

- Stores a **single character**
- Size: **1 byte**
- Stored as ASCII value internally

Example:

```
char grade = 'A';
```

---

e. void

Represents **no value** or **empty data type**.

Uses:

1. **void function** (no return value)

```
void display() {
```

```
    printf("Hello");
}
```

2. **void pointers**
    - Can hold the address of any data type.

```
void *ptr;
```

3. **void parameter**
    - Used when a function takes no arguments.

```
void hello(void);
```

---

2 Derived Data Types

These are constructed from basic data types.

---

a. Arrays

- Collection of **similar data types** stored in contiguous memory.
- Index starts from **0**.

Example:

```
int marks[5] = {90, 85, 78, 92, 88};
```

---

b. Pointers

- Store the **memory address** of another variable.

Example:

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

---

c. Functions

- Functions also have return types determined by basic or derived types.

Example:

```
int add(int x, int y) {
    return x + y;
}
```

---

d. Structures (struct)

- Used to group **different types** of data under one name.

Example:

```
struct student {
    char name[20];
    int roll;
    float marks;
};
```

Usage:

```
struct student s1 = {"Rahul", 10, 85.6};
```

---

e. Unions (union)

- Similar to structures, but all members **share the same memory location**.
- Only one member can store value at a time.

Example:

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

---

3 User-Defined Data Types

These types are created by the user.

---

a. typedef

- Used to give another name (alias) to an existing type.

Example:

```
typedef unsigned long int ULI;
ULI number = 123456;
```

---

b. enum (Enumeration)

- A set of integer constants assigned names for readability.
- Default values start at **0**, unless specified.

Example:

```
enum week {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
enum week day = Wed;   // day = 2
```

11. What is type conversion? Differentiate implicit and explicit casting.

Type conversion is the process of converting one data type into another.
In C, when the type of an expression or variable changes automatically or manually during a program, it is called type conversion.

Type conversion ensures that operations involving multiple data types are performed correctly.

There are **two types** of type conversion:

1. **Implicit Type Conversion (Automatic)**
2. **Explicit Type Conversion (Type Casting)**

Difference Between Implicit and Explicit Casting

Implicit Type Conversion	Explicit Type Conversion
Also called **automatic conversion**.	Also called **type casting**.
Done **automatically** by the compiler.	Done **manually** by the programmer.

Implicit Type Conversion	Explicit Type Conversion
Occurs when a smaller data type is converted to a larger data type (widening).	Allows conversion from larger to smaller or incompatible types (narrowing).
No risk of data loss.	Possible **data loss** during conversion.
Syntax: **No special syntax required.**	Syntax: *(data_type) expression*
Example: `int a = 5; float b = a;`	Example: `float x = 12.75; int y = (int)x;`

---

1. Implicit Type Conversion (Automatic)

Occurs when the compiler converts data types automatically.
Conversion happens from **lower** to **higher** data types:

char → int → float → double

Example:

```
int a = 10;
float b;

b = a;   // int is automatically promoted to float
```

Here, `a` becomes `10.0` automatically.

---

2. Explicit Type Conversion (Type Casting)

The programmer manually converts one data type to another using a **cast operator**.

Syntax:

```
(data_type) expression
```

Example:

```
float x = 9.89;
int y;
```

```
y = (int)x;   // float is explicitly converted to int
```

Here, `y` becomes `9` (decimal part is removed).

Final Summary

- Type conversion is changing one data type into another.
- **Implicit**: automatic, safe, widening conversion.
- **Explicit**: manual, may cause data loss, narrowing conversion.

12. Explain briefly about expression evaluation and its precedence.

Expression Evaluation in C

Expression evaluation in C refers to the process by which the compiler calculates the value of an expression.
An expression is made up of **operands** (variables, constants) and **operators** (+, –, *, /, etc.).

When evaluating an expression, C follows two important rules:

1. **Operator Precedence**
2. **Operator Associativity**

These rules decide **which operation is performed first** in a complex expression.

---

1. Operator Precedence

Operator precedence determines **priority**—which operator is evaluated before others.

For example:

```
int a = 5 + 3 * 2;
```

- `*` has higher precedence than `+`

- So expression becomes:
  `5 + (3 * 2)` = **11**

If we want to change the evaluation order, we use **parentheses**:

```
int a = (5 + 3) * 2;    // Result = 16
```

---

2. Operator Associativity

Associativity decides **evaluation direction** when two operators have the **same precedence**.

There are two types:

a) Left-to-Right Associativity

Used by:
+, -, *, /, %, <, >, <=, >=, ==, !=

Example:

```
10 - 5 - 2
```

Evaluation: `(10 - 5) - 2 = 3`

---

b) Right-to-Left Associativity

Used by:
=, +=, -=, *=, /=, ++ (post/pre), unary operators

Example:

```
a = b = c = 10;
```

Evaluation: `c = 10`, then `b = 10`, then `a = 10`

Operator Precedence Table (Simplified)

Precedence Level	Operators	Associativity

Precedence Level	Operators	Associativity
Highest	`()`, `[]`, `++`, `--`	Left to right
High	`*`, `/`, `%`	Left to right
Medium	`+`, `-`	Left to right
Low	`<`, `>`, `<=`, `>=`	Left to right
Lower	`==`, `!=`	Left to right
Very Low	`&&`	Left to right
Very Low	`` ` ``	
Lowest	`=`, `+=`, `-=`	Right to left

Brief Summary

- **Expression evaluation** is the process of computing the value of an expression.
- **Operator precedence** decides which operator is evaluated first.
- **Associativity** decides evaluation direction when operators have the same precedence.
- Parentheses can override default precedence.

UNIT – II: Input/Output, Control Statements, Arrays & Strings

---

1. Differentiate between formatted and non-formatted I/O functions

C language provides two categories of Input/Output (I/O) functions to interact with users:

1. **Formatted I/O functions**
2. **Non-formatted I/O functions**

Both perform reading and printing operations but differ in how they handle data.

1. Formatted I/O Functions (Detailed Explanation)

Definition

Formatted I/O functions allow the programmer to **control the format** in which data is read or displayed.
These functions use **format specifiers** to define the type, width, and precision of the data.

Common Functions

- `printf()` → formatted output
- `scanf()` → formatted input

Format Specifiers

Used to identify data types:

- `%d` → integer
- `%f` → float
- `%c` → character
- `%s` → string
- `%lf` → double

Features

✓ Handles multiple data types

✓ Allows formatting output (width, alignment, precision)

✓ Automatically converts input/output data based on format specifier

✓ Supports escape sequences (`\n`, `\t`)

✓ Suitable for structured, well-presented output

Example

```
int age;
float marks;

printf("Enter age and marks: ");
scanf("%d %f", &age, &marks);

printf("Age = %d, Marks = %.2f", age, marks);
```

---

2. Non-Formatted I/O Functions (Detailed Explanation)

Definition

Non-formatted I/O functions deal directly with **characters and strings** without format control. They do not use format specifiers and do not convert data types automatically.

Common Functions

- **Character I/O:**
  - `getchar()` — read a character
  - `putchar()` — write a character
- **String I/O:**
  - `gets()` — read a string (unsafe, outdated)
  - `puts()` — write a string

Features

✓ Simple and fast (no format processing)

✓ Suitable for reading characters or entire strings

✓ No control over precision, width, alignment

✓ Cannot directly read numeric values (must be converted manually)

Example

```
char ch;
char name[20];

printf("Enter a character: ");
```

```
ch = getchar();

printf("Enter your name: ");
gets(name);

putchar(ch);
puts(name);
```

Final Summary

- **Formatted I/O functions** allow detailed control over how data is read and displayed using format specifiers.
- **Non-formatted I/O functions** handle characters and strings directly without formatting.
- Formatted I/O is powerful for structured programs, while non-formatted I/O is simple and fast for basic character/string operations.

---

2. Explain the use of printf() and scanf() with examples

C language uses standard input/output (I/O) functions to interact with the user.
Two of the most important functions are:

- **`printf()`** → for displaying/outputting data
- **`scanf()`** → for reading/inputting data

Both functions are provided by the **`<stdio.h>`** header file.

`printf()` Function — Detailed Explanation

Definition

`printf()` is a *formatted output function* used to display text, variables, and results on the screen.

Syntax

```
printf("format string", argument_list);
```

Components

1. **Format string** – contains normal text + format specifiers
2. **Argument list** – variables whose values will be printed

Format Specifiers Used in printf()

Specifier	Meaning
`%d`	Integer
`%f`	Float
`%.2f`	Float with 2 decimal places
`%c`	Character
`%s`	String
`%lf`	Double
`%u`	Unsigned integer

Examples

Example 1: Printing Text

```
printf("Welcome to C programming!");
```

Example 2: Printing Variables

```
int age = 18;
float marks = 92.5;

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

Example 3: Formatting Output

```
printf("Value = %5d", 25);
```

- Prints the number in **5 spaces** (right aligned).

`scanf()` Function — Detailed Explanation

Definition

`scanf()` is a *formatted input function* used to read values entered by the user.

Syntax

```
scanf("format string", &variable_list);
```

Important Notes

1. Uses **format specifiers** to know which data type to read.
2. Variables **must be passed with the address operator (&)** except strings.
3. Reads input **from the keyboard**.

Format Specifiers Used in scanf()

Specifier	Meaning
`%d`	Reads integer
`%f`	Reads float
`%c`	Reads character
`%s`	Reads string
`%lf`	Reads double

---

3. What are escape sequences? List any 5 with meanings.

Escape sequences are **special character combinations** in C that begin with a **backslash (\)** and are used to represent characters that cannot be typed directly on the keyboard or have special meaning in output formatting.

They are used inside **strings and character constants** to control the format of output.

Example:

```
printf("Hello\nWorld");
```

Here, `\n` is an escape sequence that moves the cursor to the next line.

✅ List of Escape Sequences with Meanings

Escape Sequence	Meaning
`\n`	Moves the cursor to a **new line** (line break)
`\t`	Inserts a **horizontal tab**
`\\`	Prints a **backslash** character
`\'`	Prints a **single quote**
`\"`	Prints a **double quote**
`\r`	Carriage return (moves cursor to beginning of line)
`\b`	Backspace
`\0`	Null character
`\a`	Alert (beep sound)

4. Write a C program to check if a number is prime.

```
#include <stdio.h>

int main() {
    int num, i, isPrime = 1;

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

    // Handle numbers less than 2
    if (num < 2) {
        isPrime = 0;
    } else {
        // Check divisibility from 2 to num/2
        for (i = 2; i <= num / 2; i++) {
            if (num % i == 0) {
                isPrime = 0;
                break;
            }
        }
    }
```

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

    return 0;
}
```

5.Write a C program to find the sum and average of an array

```
#include <stdio.h>

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

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

    int arr[n];
```

```
    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
        sum += arr[i];
    }

    average = sum / n;

    printf("Sum = %.2f\n", sum);
    printf("Average = %.2f\n", average);

    return 0;
}
```

Explanation:

The user enters the number of elements.

Elements are stored in an array.

A loop computes the sum.

The average is calculated as sum / n.

1. `if` Statement (Simple Decision Making)

The **if** statement tests a condition.
If the condition evaluates to **true**, the associated block of code is executed.
If the condition is **false**, the code block is skipped.

Syntax:

```
if (condition) {
    // Code executed only if condition is true
}
```

How it works:

- The condition inside parentheses is evaluated.
- Conditions often use **relational** (>, <, ==, >=, !=) and **logical** operators (&&, ||, !).
- Only one block executes if the condition is true.

Example:

```
if (temperature > 35) {
    printf("It's a hot day.");
}
```

Explanation:

If `temperature > 35`, the message will be printed; otherwise nothing happens.

⋆ 2. `if-else` Statement (Two-Way Decision)

This statement provides **two choices**:

- One block executes if the condition is **true**.
- Another block executes if the condition is **false**.

Syntax:

```
if (condition) {
    // Code for true condition
} else {
    // Code for false condition
}
```

Example:

```
if (age >= 18) {
    printf("You are eligible to vote.");
} else {
    printf("You are not eligible to vote.");
}
```

Explanation:

Either the `if` block or the `else` block will run—never both.

⭑ 3. Nested `if-else` Statement (Multi-Way Decision)

A **nested if** means writing one `if` or `else` block **inside another**.
It is used when you need to check **multiple conditions** in sequence.

Syntax:

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

Example (Student grading system):

```
if (marks >= 90) {
    printf("Grade A");
} else if (marks >= 75) {
    printf("Grade B");
} else if (marks >= 50) {
    printf("Grade C");
} else {
    printf("Fail");
}
```

Explanation:

- Conditions are checked **one by one**.
- First true condition's block executes.
- Remaining conditions are skipped.

When to use nested if-else:

- When decisions are **dependent**.
- When there are **ranges** or **multiple conditions**.

⭐ 4. `switch` Statement (Multi-Way Menu Style Decision)

The `switch` statement evaluates an expression and compares it with different **case values**.
It is often used when checking **one variable** against many constant values.

Syntax:

```
switch (expression) {
    case value1:
        // statements
        break;

    case value2:
        // statements
        break;

    ...

    default:
        // statements if no case matches
}
```

Example (Day of week):

```
switch (day) {
    case 1:
        printf("Monday");
        break;
    case 2:
        printf("Tuesday");
        break;
    case 3:
        printf("Wednesday");
        break;
    default:
        printf("Invalid day");
}
```

7. Explain selection control statements: if, if-else, nested if-else, switch.

1. `if` Statement (Simple Decision Making)

The **if** statement tests a condition.
If the condition evaluates to **true**, the associated block of code is executed.
If the condition is **false**, the code block is skipped.

Syntax:

```
if (condition) {
    // Code executed only if condition is true
}
```

How it works:

- The condition inside parentheses is evaluated.
- Conditions often use **relational** (>, <, ==, >=, !=) and **logical** operators (&&, ||, !).
- Only one block executes if the condition is true.

Example:

```
if (temperature > 35) {
    printf("It's a hot day.");
}
```

Explanation:

If `temperature > 35`, the message will be printed; otherwise nothing happens.

⭐ 2. `if-else` Statement (Two-Way Decision)

This statement provides **two choices**:

- One block executes if the condition is **true**.
- Another block executes if the condition is **false**.

Syntax:

```
if (condition) {
    // Code for true condition
} else {
    // Code for false condition
```

```
}
```

Example:

```
if (age >= 18) {
    printf("You are eligible to vote.");
} else {
    printf("You are not eligible to vote.");
}
```

Explanation:

Either the `if` block or the `else` block will run—never both.

⭐ 3. Nested `if-else` Statement (Multi-Way Decision)

A **nested if** means writing one `if` or `else` block **inside another**.
It is used when you need to check **multiple conditions** in sequence.

Syntax:

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

Example (Student grading system):

```
if (marks >= 90) {
    printf("Grade A");
} else if (marks >= 75) {
    printf("Grade B");
} else if (marks >= 50) {
    printf("Grade C");
} else {
    printf("Fail");
}
```

Explanation:

- Conditions are checked **one by one**.

- First true condition's block executes.
- Remaining conditions are skipped.

When to use nested if-else:

- When decisions are **dependent**.
- When there are **ranges** or **multiple conditions**.

⭐ 4. `switch` Statement (Multi-Way Menu Style Decision)

The `switch` statement evaluates an expression and compares it with different **case values**.
It is often used when checking **one variable** against many constant values.

Syntax:

```
switch (expression) {
    case value1:
        // statements
        break;

    case value2:
        // statements
        break;

    ...

    default:
        // statements if no case matches
}
```

Example (Day of week):

```
switch (day) {
    case 1:
        printf("Monday");
        break;
    case 2:
        printf("Tuesday");
        break;
    case 3:
        printf("Wednesday");
        break;
    default:
        printf("Invalid day");
}
```

8. Explain while, do-while, and for loops with flowcharts and examples

What is a Loop?

A **loop** in programming is a control structure that allows a set of statements to be executed **repeatedly** as long as a specified **condition** remains true.

✓ Why do we use loops?

- To avoid writing repetitive code
- To simplify tasks like printing numbers, processing arrays, reading input repeatedly, etc.

✓ Types of loops in C:

1. **while loop**
2. **do-while loop**
3. **for loop**

⋆ 1. while Loop

A **while loop** is an **entry-controlled loop**, meaning the condition is checked **before** the loop body executes.

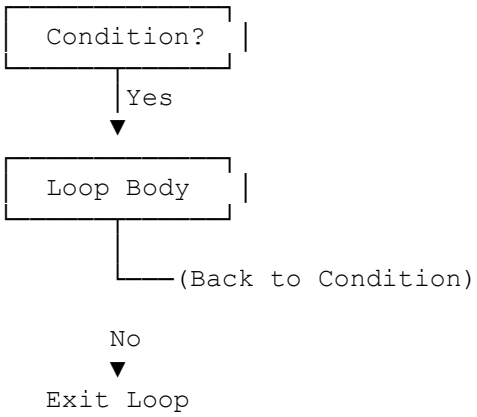
✓ Syntax

```
while (condition) {
    // loop body
}
```

✓ How it works

1. Condition is checked.
2. If TRUE → loop body executes.
3. If FALSE → loop terminates.
4. After executing body, control returns to check the condition again.

✓ Flowchart



✓ Example: Print numbers 1 to 5

```
#include <stdio.h>

int main() {
    int i = 1;

    while (i <= 5) {
        printf("%d ", i);
        i++;
    }
    return 0;
}
```

⋆ 2. do-while Loop

A **do-while loop** is an **exit-controlled loop**.
The loop body executes **at least once**, even if the condition is false.

✓ Syntax

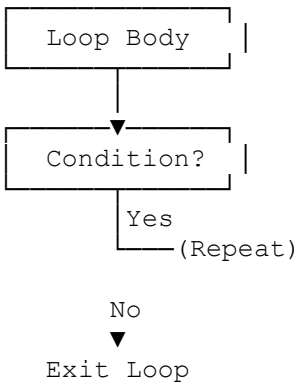
```
do {
    // loop body
} while (condition);
```

✓ How it works

1. Loop body executes first.
2. Condition is checked afterward.
3. If TRUE → loop repeats.

4. If FALSE → loop stops.

✓ Flowchart



✓ Example: Print numbers 1 to 5

```
#include <stdio.h>

int main() {
    int i = 1;

    do {
        printf("%d ", i);
        i++;
    } while (i <= 5);

    return 0;
}
```

⋆ 3. for Loop

A **for loop** is used when the number of iterations is **known in advance**.
It combines initialization, condition checking, and increment/decrement in one line.

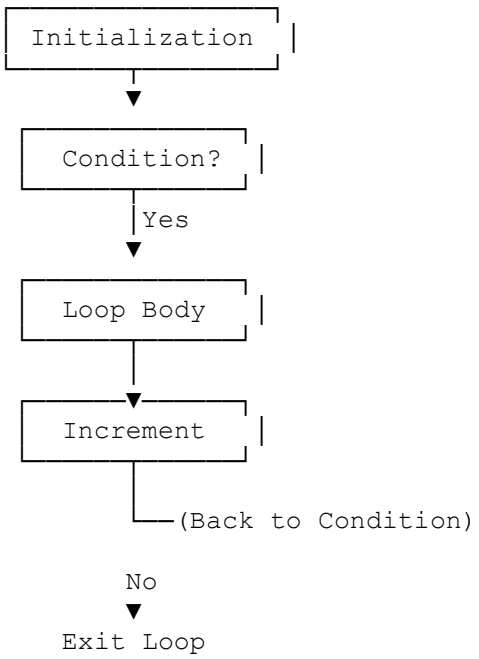
✓ Syntax

```
for (initialization; condition; increment/decrement) {
    // loop body
}
```

✓ How it works

1. Initialization runs once.
2. Condition is checked:
    - TRUE → execute loop body
    - FALSE → exit loop
3. After each iteration, increment/decrement is executed.

✓ Flowchart



9. Differentiate between break, continue, and goto statements.

Jump statements in C are statements that **transfer program control** from one part of the program to another.
They allow the program to skip certain instructions, exit loops early, or jump to a labeled location.

✓ Types of Jump Statements in C:

1. **break**
2. **continue**
3. **goto**

4. **return** (also a jump statement, used to exit a function)

In this question, we focus on **break**, **continue**, and **goto**.

⋆ 1. `break` Statement

✓ Purpose:

To **terminate** a loop or exit a `switch` block immediately.

✓ Used in:

- `for` loop
- `while` loop
- `do-while` loop
- `switch` statement

✓ How it behaves:

When `break` is executed, control jumps **out of the loop or switch**, and continues with the next statement after it.

✓ Example:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3)
        break;
    printf("%d ", i);
}
```

Output:
```
1 2
```

⋆ 2. `continue` Statement

✓ Purpose:

To **skip the remaining statements of the current iteration** and move to the **next iteration** of the loop.

✓ Used in:

- `for` loop
- `while` loop
- `do-while` loop

✓ How it behaves:

- In a `for` loop → jumps to increment/decrement step
- In `while` / `do-while` → jumps to condition checking

✓ Example:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3)
        continue;
    printf("%d ", i);
}
```

Output:
`1 2 4 5`
(Iteration when `i = 3` is skipped)

⋆ 3. `goto` Statement

✓ Purpose:

To **transfer control unconditionally** to another point in the program using a label.

✓ Used in:

- Anywhere inside a function

✓ How it behaves:

Program control jumps directly to the labeled statement, regardless of program flow.

✓ Example:

```
#include <stdio.h>

int main() {
    int i = 1;

start:
    printf("%d ", i);
    i++;

    if (i <= 5)
        goto start;

    return 0;
}
```

Output:

```
1 2 3 4 5
```

10. What is an array? Explain about types of arrays with examples?

What is an Array?

An **array** is a **collection of elements of the same data type** stored in **contiguous memory locations**. Arrays allow you to store multiple values under a **single variable name** and access them using an **index**.

- **Key Points:**
  - All elements must be of the **same data type**.
  - Indexing usually starts from **0**.
  - Arrays can be **one-dimensional or multi-dimensional**.

Example in C:

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

Here:

- `numbers` is the array name.
- `int` is the data type.
- `5` is the size of the array.
- Elements can be accessed as `numbers[0]` (10), `numbers[1]` (20), etc.

---

2. Types of Arrays

Arrays can be classified based on **dimensions**:

A. One-Dimensional Array (1D Array)

- Stores elements in a **single row** or a **single column**.
- Accessed by a **single index**.

Example:

```
int arr[4] = {5, 10, 15, 20};
printf("%d", arr[2]); // Output: 15
```

B. Two-Dimensional Array (2D Array)

- Stores data in **rows and columns**, like a **matrix**.
- Accessed by **two indices**: `array[row][column]`.

Example:

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

Memory Layout:

```
matrix[0][0] matrix[0][1] matrix[0][2]
matrix[1][0] matrix[1][1] matrix[1][2]
```

C. Multi-Dimensional Array (3D or more)

- Arrays with **more than two dimensions**.
- Used for complex data representation, like **3D graphics, scientific computations**.
- Accessed by **multiple indices**.

Example (3D Array):

```
int cube[2][2][2] = {
  {{1,2}, {3,4}},
  {{5,6}, {7,8}}
};
printf("%d", cube[1][0][1]); // Output: 6
```

D. Dynamic Arrays (in some languages like C++, Python)

- Size can **change during runtime**.
- Example in **Python**:

```
arr = [1, 2, 3]
arr.append(4)  # arr becomes [1, 2, 3, 4]
```

1. Character Arrays

A **character array** is an array that stores **characters**. It is commonly used to store **strings** in C.

- **Definition:**

```
char name[10];  // Can store up to 9 characters + 1 null character '\0'
```

- **Important:** Strings in C are **terminated with a null character `\0`**. This marks the **end of the string**.

Example:

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
printf("%s", str);  // Output: Hello
```

Or more commonly:

```
char str[] = "Hello"; // Compiler automatically adds '\0'
```

---

2. String Functions from `<string.h>`

C provides a header file called **`string.h`** which contains useful functions for **manipulating strings**. Let's discuss the ones you mentioned.

A. strlen() – String Length

- Returns the **length of the string** (number of characters **excluding** `\0`).

Syntax:

```
#include <string.h>
size_t strlen(const char *str);
```

Example:

```
char str[] = "Hello";
printf("%lu", strlen(str));  // Output: 5
```

B. strcpy() – String Copy

- Copies **source string** into **destination string**.

Syntax:

```
#include <string.h>
char *strcpy(char *dest, const char *src);
```

Example:

```
char src[] = "Hello";
char dest[10];
strcpy(dest, src);
printf("%s", dest);  // Output: Hello
```

Important: Make sure `dest` is **large enough** to hold the source string + `\0`.

C. strcmp() – String Compare

- Compares two strings **lexicographically** (dictionary order).
- Returns:
  - `0` → strings are equal
  - `<0` → first string is **less than** second
  - `>0` → first string is **greater than** second

Syntax:

```
#include <string.h>
int strcmp(const char *str1, const char *str2);
```

Example:

```
char str1[] = "Hello";
char str2[] = "World";
printf("%d", strcmp(str1, str2));  // Output: negative number
```

D. strcat() – String Concatenation

- Appends the **source string** to the **end of the destination string**.

Syntax:

```
#include <string.h>
char *strcat(char *dest, const char *src);
```

Example:

```
char str1[20] = "Hello ";
char str2[] = "World";
strcat(str1, str2);
printf("%s", str1);  // Output: Hello World
```

Important: `str1` must have **enough space** to hold both strings and the null character.

11. Explain character arrays and string functions (strlen, strcpy, strcmp, strcat) from string.h.

1. Character Arrays

A **character array** is an array that stores **characters**. It is commonly used to store **strings** in C.

- **Definition:**

```
char name[10];  // Can store up to 9 characters + 1 null character '\0'
```

- **Important:** Strings in C are **terminated with a null character `\0`**. This marks the **end of the string**.

Example:

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
printf("%s", str);  // Output: Hello
```

Or more commonly:

```
char str[] = "Hello"; // Compiler automatically adds '\0'
```

---

2. String Functions from `<string.h>`

C provides a header file called **`string.h`** which contains useful functions for **manipulating strings**. Let's discuss the ones you mentioned.

A. strlen() – String Length

- Returns the **length of the string** (number of characters **excluding** `\0`).

Syntax:

```
#include <string.h>
size_t strlen(const char *str);
```

Example:

```
char str[] = "Hello";
printf("%lu", strlen(str));  // Output: 5
```

B. strcpy() – String Copy

- Copies **source string** into **destination string**.

Syntax:

```
#include <string.h>
char *strcpy(char *dest, const char *src);
```

Example:

```
char src[] = "Hello";
char dest[10];
strcpy(dest, src);
printf("%s", dest);  // Output: Hello
```

Important: Make sure `dest` is **large enough** to hold the source string + `\0`.

C. strcmp() – String Compare

- Compares two strings **lexicographically** (dictionary order).
- Returns:
  - `0` → strings are equal
  - `<0` → first string is **less than** second
  - `>0` → first string is **greater than** second

Syntax:

```
#include <string.h>
int strcmp(const char *str1, const char *str2);
```

Example:

```
char str1[] = "Hello";
char str2[] = "World";
printf("%d", strcmp(str1, str2));  // Output: negative number
```

D. strcat() – String Concatenation

- Appends the **source string** to the **end of the destination string**.

Syntax:

```
#include <string.h>
char *strcat(char *dest, const char *src);
```

Example:

```
char str1[20] = "Hello ";
char str2[] = "World";
strcat(str1, str2);
printf("%s", str1);  // Output: Hello World
```

Important: `str1` must have **enough space** to hold both strings and the null character.